MIA-Generation – User Guide

- Tutorial -

Mia-Software
model-driven agility

Table of contents

TUTORIAL **1**

PRESENTATION 1
FIRST GENERATION OF JAVA CODE 2
Creating a generation project *3*
Creating a package *4*
Creating a file template *5*
Saving a script and its package *8*
Loading a model *9*
Importing a model from a CASE tool 9
Testing a file template *12*
Creating a generation unit 12
Generating a file template 16
Filename property 18
IMPROVING THE JAVA GENERATION 19
Inheritance Generation *19*
Creation of a superClasses service 19
Creating a superClassName macro 24
Call the macro superClassName 25
Generating trace *26*
Attribute Generation *28*
Adding a template for attribute declaration 28
Adding a template to generate the accessors 30
Generation of the attributes from a class 31
Creating a macro with parameters 34
Parameters naming 34
Parameters type 34
Calling a script with parameters in Java scripts 36
Calling a script with parameters in a template 37
User Code Management *38*
Filename resolution 42
GENERATION BY SCENARIO 45
Creating a scenario *45*
Using services into scenarios *48*
Getting all classes 48
Calling allClasses service into a scenario 50
ADVANCED CONCEPTS 51
Using profile *58*
Creation of a profile 59
Creating scripts on the profile types 60
Reorganizing scripts 60
Coding conventions, patterns and frameworks *65*
Coding conventions and patterns 65
Frameworks 71
Using J2SE components into scripts *73*
Context variable *75*
MIA-Generation Javadoc *77*

Tutorial

Presentation

The goal of this tutorial is not to present all of the functions available MIA-Generation, but to introduce you to this working environment, and to have you discover with the aid of a simple example of Java code generation, the fundamental principles of generation with MIA-Generation.

The example presented in this tutorial utilizes MIA-Generation for UML.

All scripts defined in this tutorial can be found in "Tutorial …" projects in the directory :

```
<MIA-Generation>\metamodels\UML\examples\generators
```

The project "`Tutorial (scenario generation).prj`" references scritps from "First Generation of Java Code" to "*Scenario Generation"* chapters. Scripts from "*Advanced concepts"* chapter can be loaded with "`Tutorial (with stereotype).prj`", "`Tutorial (with profile).prj`", "`Tutorial (with profile and pattern).prj`" projects.

All concepts from this tutorial are detailed in MIA-Generation manual. A technical FAQ is also avalaible.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

First Generation of Java Code

Start MIA-Generation Architect from the Windows "Start" Menu.MIA-Generation Architect is an application that allows you to build a specific generator for a given context. A generation project is a grouping of everything that is necessary for a given generation.

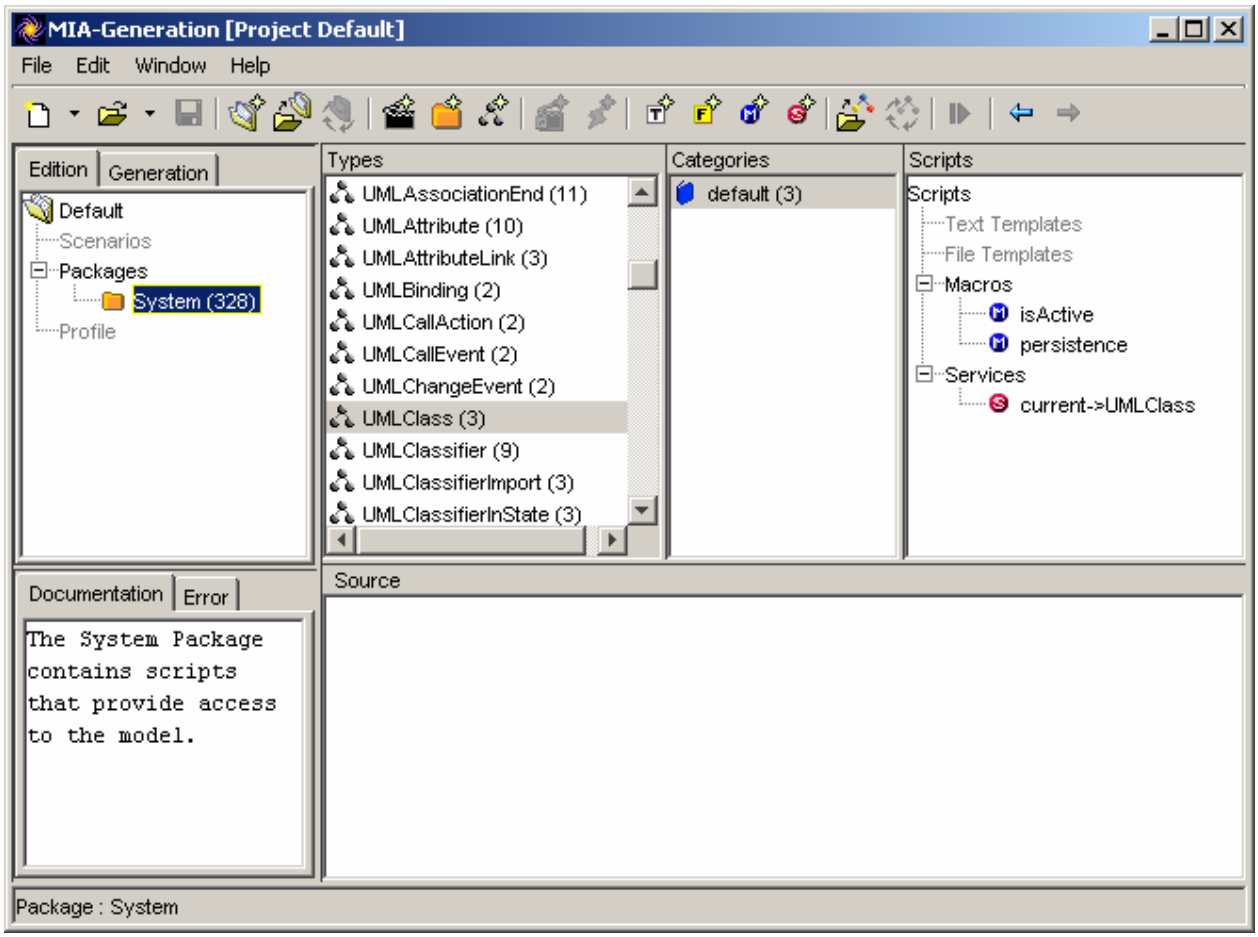


Figure 1 - MIA-Generation Architect

MIA-Generation Architect provides two main panels :

- The **Edition** panel provides you with creation, saving and edition functionnality. For each element shown in the Navigator, you can open a specific editor showing the corresponding element.
- The **Generation** panel provides you with generation and trace functionnality.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating a generation project

A generation project is a grouping of everything that is necessary for a given generation. A generation project can contain packages, scenarios and a profile. A project defines a generator based on scripts defined for a meta-model. A project allows you to save the entire work environment, which can be restored at a later time.

There is always a current project in MIA-Generation. When the application is opened, an empty project ("Default") is created by default.

To create a new project, you can click the New Project button or use the New button from the toolbar.

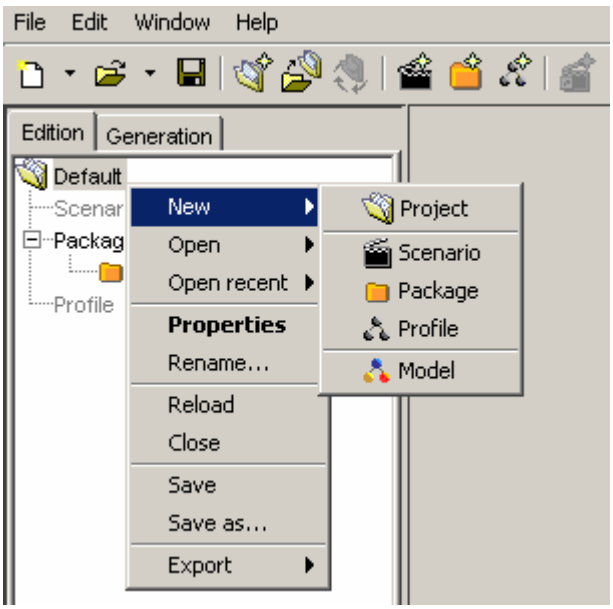


Figure 2 – New > Project menu

Specify the name of the new project :

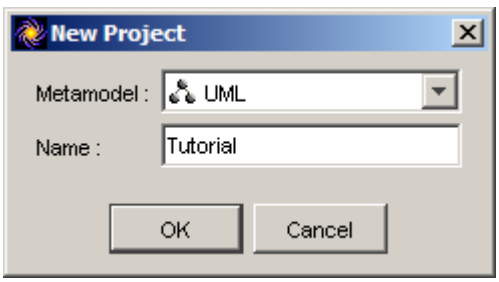


Figure 3 – Creation of a Project

The project navigator now shows you the *Tutorial* project that you created.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating a package

Each generation script must be attached to a **package**. The purpose is to be able to organize, in different entities, the scripts written for different requirements.

However, the only one present when MIA-Generation is started is the package "System" which cannot be modified. This package "System" contains all the base scripts for accessing the loaded model.

To create a package named *Tutorial*, click the New Package button from the toolbar and specify the name for the new package:

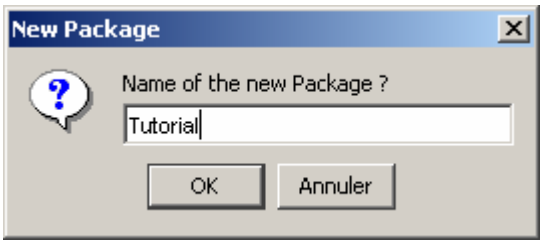


Figure 4 – Creation of a package

The project navigator now shows two packages : the package "System" and your newly created package.

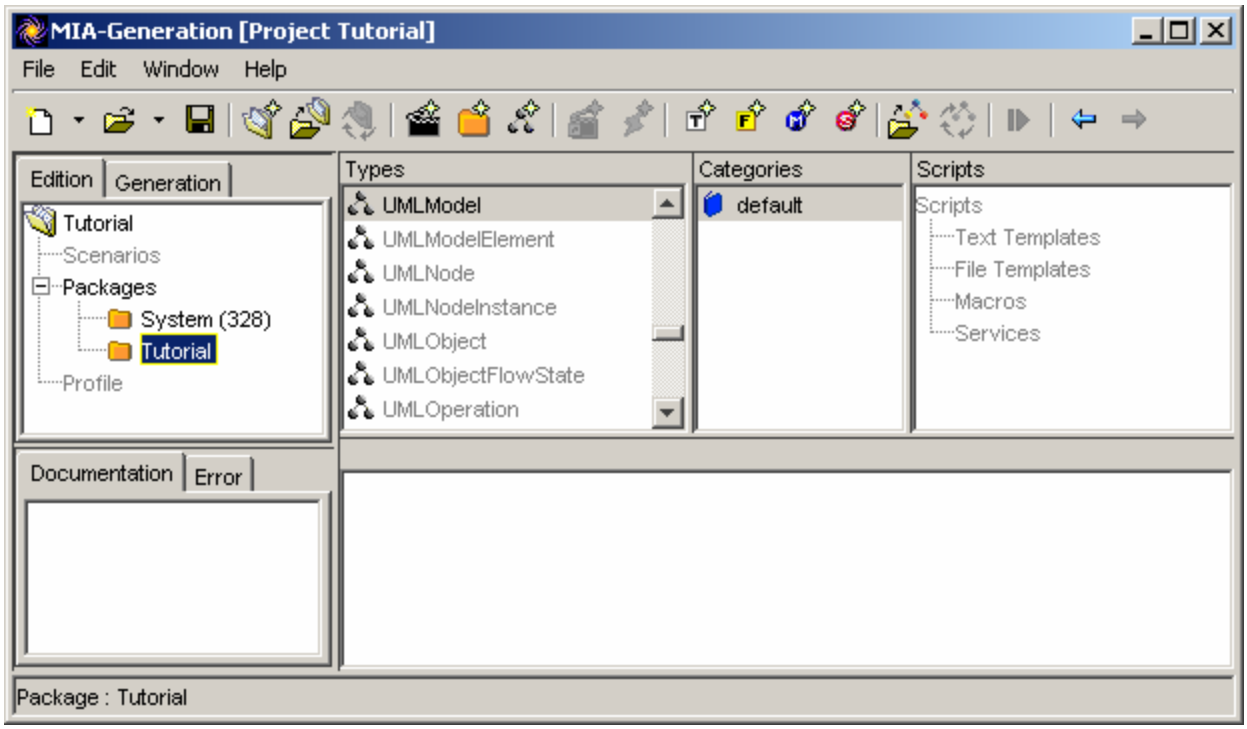


Figure 5 – Project Navigator

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating a file template

We are going to write a simple generation template to create a Java file for each class in the model.

A template is text, which describes in a visual manner, the way in which the code must be generated for an object type of the model.The text of a template is composed of **textual elements** which will be reproduced just as they are in the generated text and **scripts names** which, at the time of evaluation, will be inserted in the generated text.

It is possible to specify the filename where the result of a template must be saved. To do that, it is necessary to create a **File template** of it. It is a template to which another template **Filename** is systematically connected. Once evaluated, this template **Filename** gives the name of the file in which the text of the file template will be saved.

To create a *File Template* :

1. Select the package, the type and the category where you want to create the script.
2. Click the New File Template button from the toolbar or use **New > File Template** from the Scripts list context menu.
3. Specify the name of the new file template : *javaGeneration*.

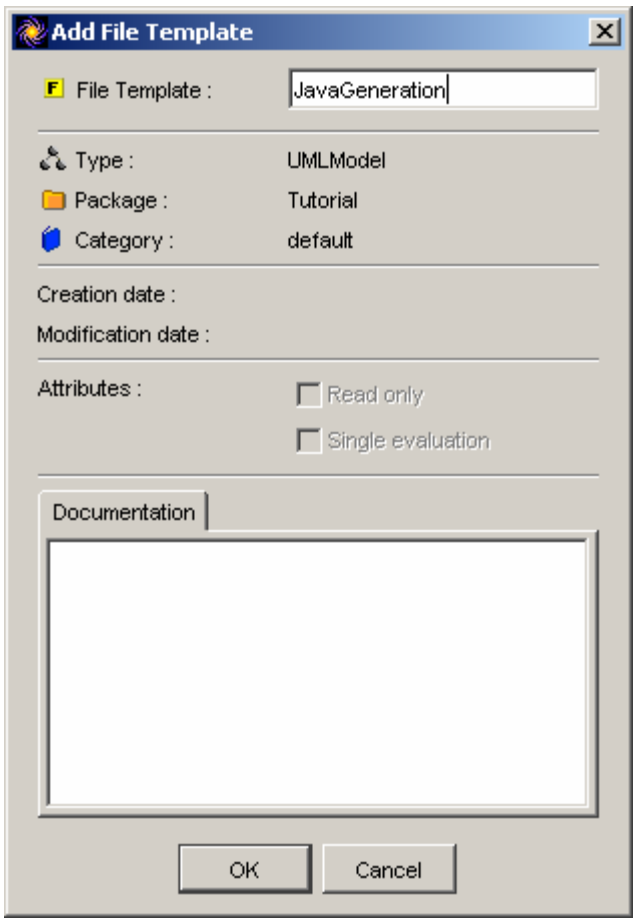


Figure 6 - File Template properties

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The **Scripts** list now shows you the File Template that you created, with the text "generated text" created by default.

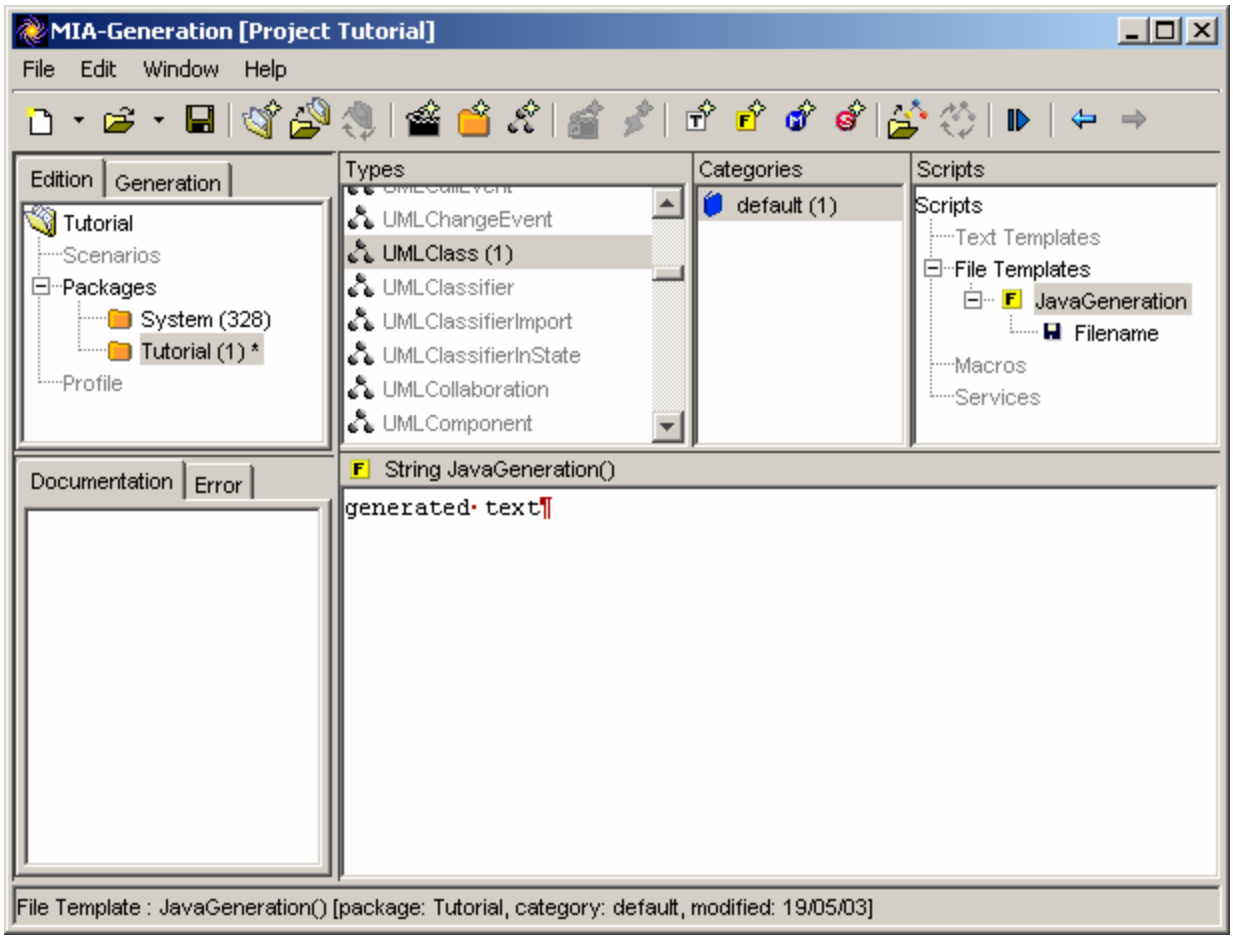


Figure 7 – Scripts list and Script edition area

A template allows you to specify in a manner WYSIWYG (without programming) the text that must be produced. We are going to see how to modifiy this template to generate Java™ class body for each UML class from the model.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Modify the text of this template. Validate your modifications with the context-sensitive menu "Save" in this textbox, or by using the keyboard shortcut CTRL+S.

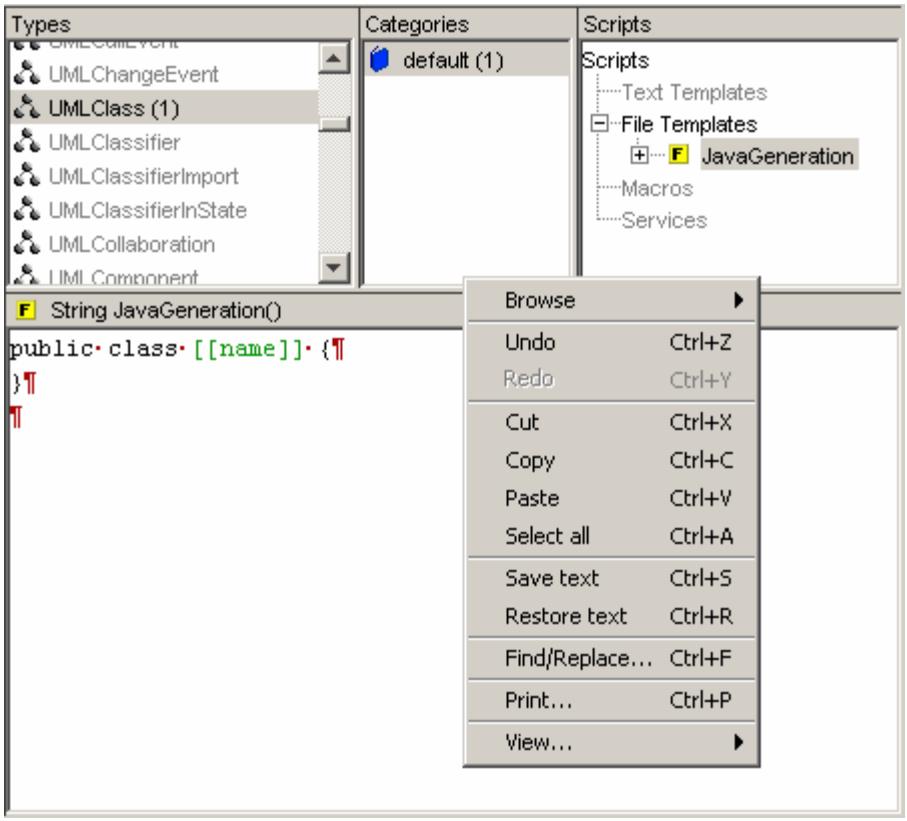


Figure 8 – Changing the generated text

Scripts describe how text can be generated from a model. There is two types of scripts :

- Generation models or **templates** : a template is text, which describes in a visual manner, the way in which the code must be generated for an object type of the model. The script editor is WYSIWYG (What You See Is What You Get).
- **Java™ scripts** : Because everything cannot be expressed as WYSIWYG, scripts written in Java are used. There are two kinds of JAVA™ scripts : **macros** which are evaluated for a model object and whose result is text or JAVA™ objects and **services** wich are special macros that enable you to return objects from your model.

The *name* script is a Java™ script, a macro which returns a String that represents the name of the current class. This macro is defined in the "System" package (which contains all the base scripts for accessing the loaded model).

To distinguish scripts names, they are framed by special characters (**[[** and **]]** by default), which can be modified according to generation requirements (see the user manual Preferences chapter).

This template will now generate the text "public class ", followed by the name of the class of the loaded model within which this script will be evaluated (call of the script "name" defined in the “System” package), followed by two curly braces "`{ }`".

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Saving a script and its package

When you modify the text of a script, you can validate it using with **CTRL+S** shorcut or **Save Text** context menu from edition area. Text is compiled but not saved into file : **saving a script is carried out by means of saving the package in which it is contained.** To save a package and scripts it contains (a package must be selected), click the Save button from the toolbar or use **File > Save <**Package> from the main menu bar.

Note : For each saving of a package, a backup file is created from the old file (renamed by adding the extension ".bak").

Remember to save your package by using the Save button from the toolbar.

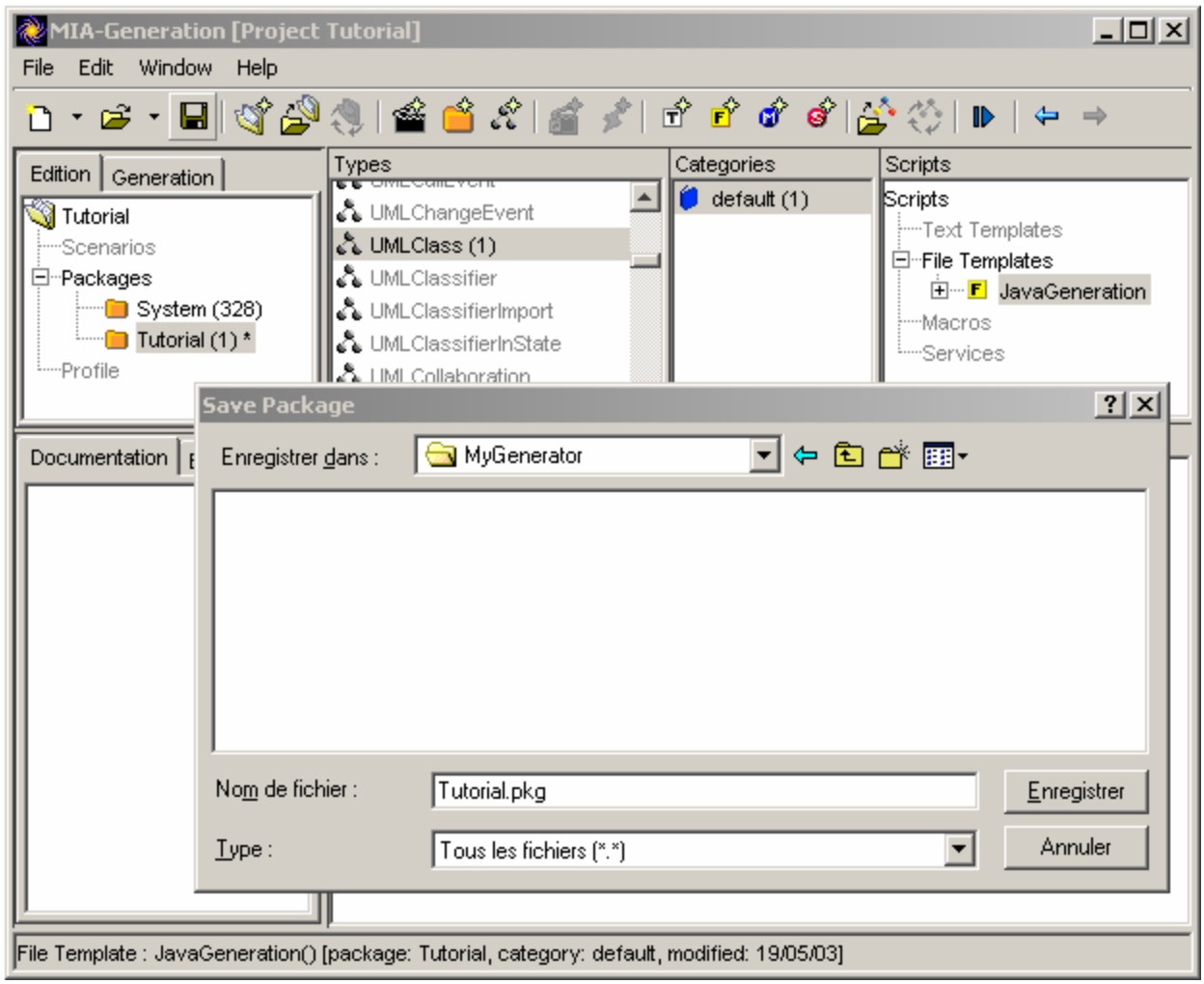


Figure 9 – Saving a package

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Loading a model

The models used for generation are managed from the **Generation** panel. MIA-Generation only manages one model at a time. This model can be imported from a CASE tool.

Importing a model from a CASE tool

You can import a model created with a CASE tool using the Load button and the Model item Model from the toolbar.

MIA-Generation is capable of importing models from many tools. Therefore you must choose the type of model to import :

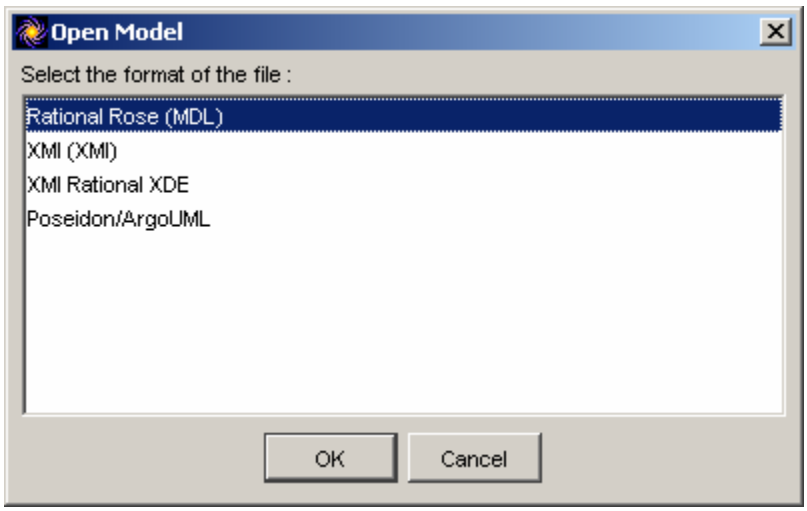


Figure 10 - Model formats

Once you have chosen the format, you can select the file to analyze. Choose "Rational Rose" for the import format and open the `Tutorial.mdl` file :

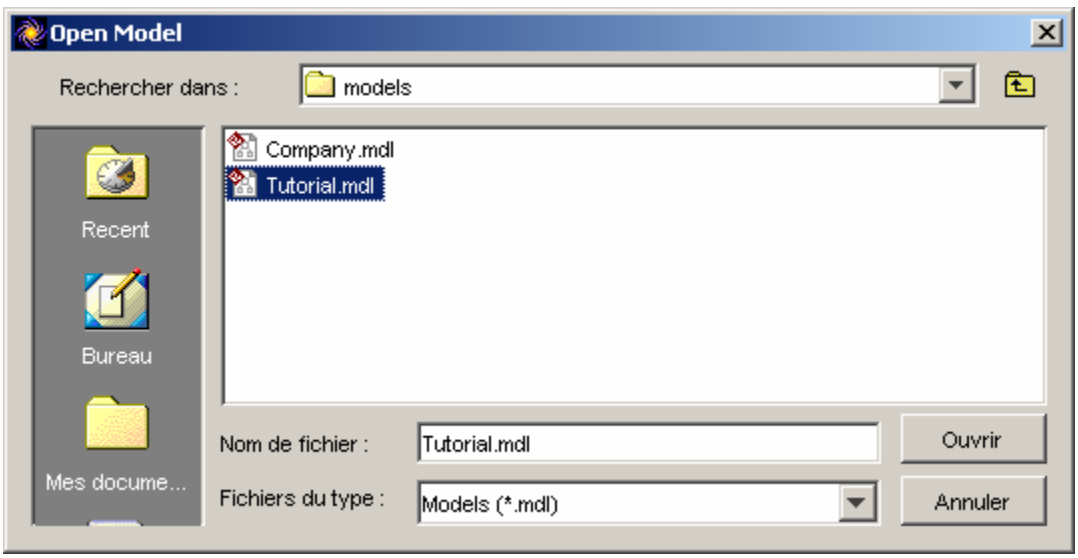


Figure 11 - Selection of the model to import

This file is located in the directory `<MIA-Generation>\metamodels\UML\examples\models`. The imported model then replaces the current model in MIA-Generation.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The model can be viewed using the browser integrated into MIA-Generation. This browser allows navigating through all the objects of the model. And for each object, its properties and attached objects can be viewed.

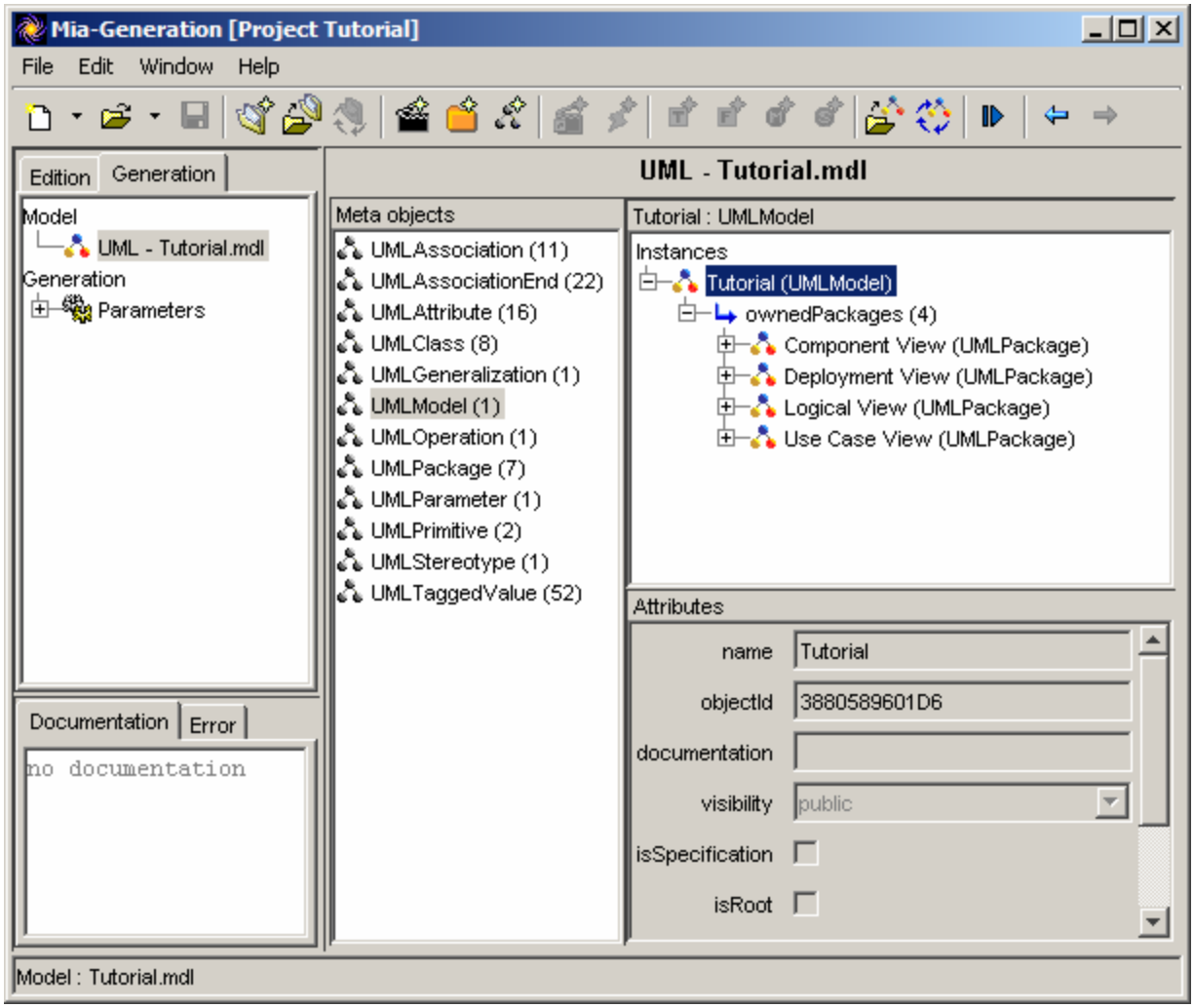


Figure 12 – Model browser

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Once the model is loaded, MIA-Generation opens the model browser on the principle element : the object *UMLModel*.

While expanding the instance tree, you can view that this model contains four packages, that the "Logical View" package contains a "Building" sub-package, that this package contains an "Address" class ,etc.

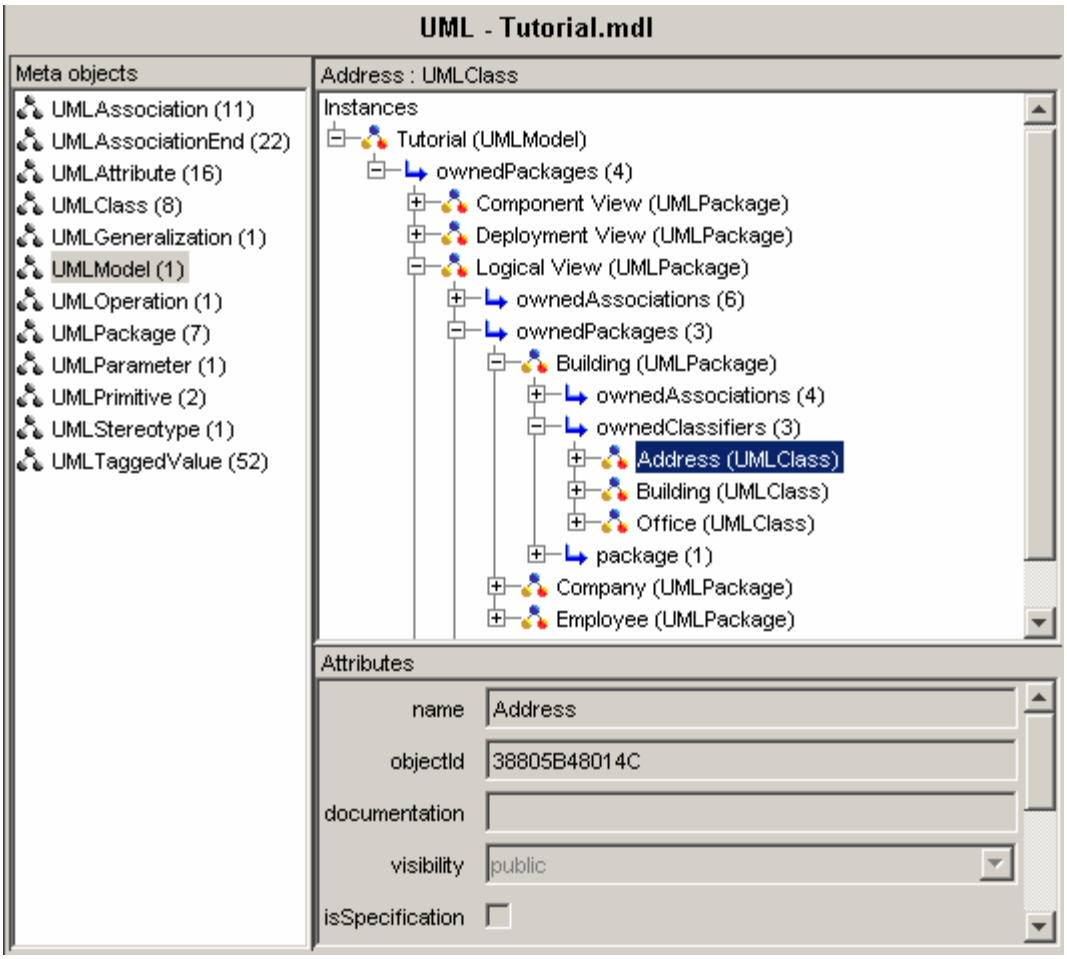


Figure 13 – Model browser

You can continue this process and traverse through all the objects in the model : the packages contain classes, which contain attributes, etc…

Browsing through the model allows you to see how the model is loaded in the application, including the links between the different objects, which will make it easier to write the generation scripts.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Testing a file template

We are going to create a generation unit to test the *JavaGeneration* file template. Generation breaks down into three steps. Initially, it is necessary to select scripts you want to test. Then, the objects to which the generation will apply must be chosen. Then, you just have to launch the generation.

Creating a generation unit

To create a generation unit, use **Generate…** item from the Scripts list context menu or the Generate from the toolbar if you are on the **Edition** panel.

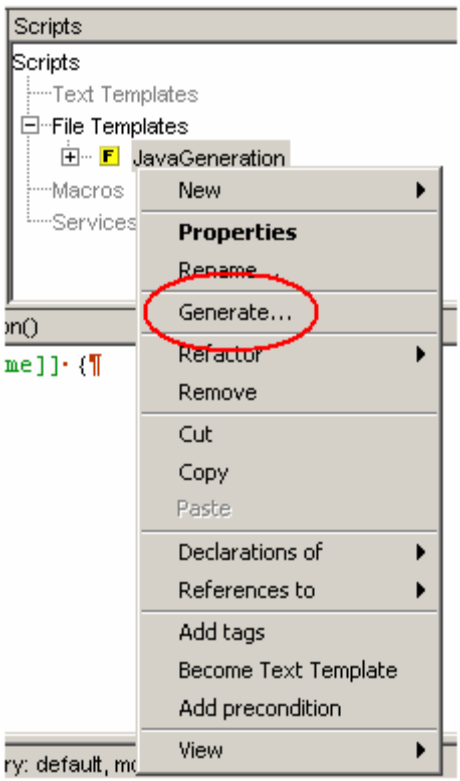


Figure 14 – Scripts list context menu

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

First, a generation unit named <*Script – Type*> is created and you just need to select objects to which the generation will apply. Then, if you click **Generate…** item, generation unit will be launch.

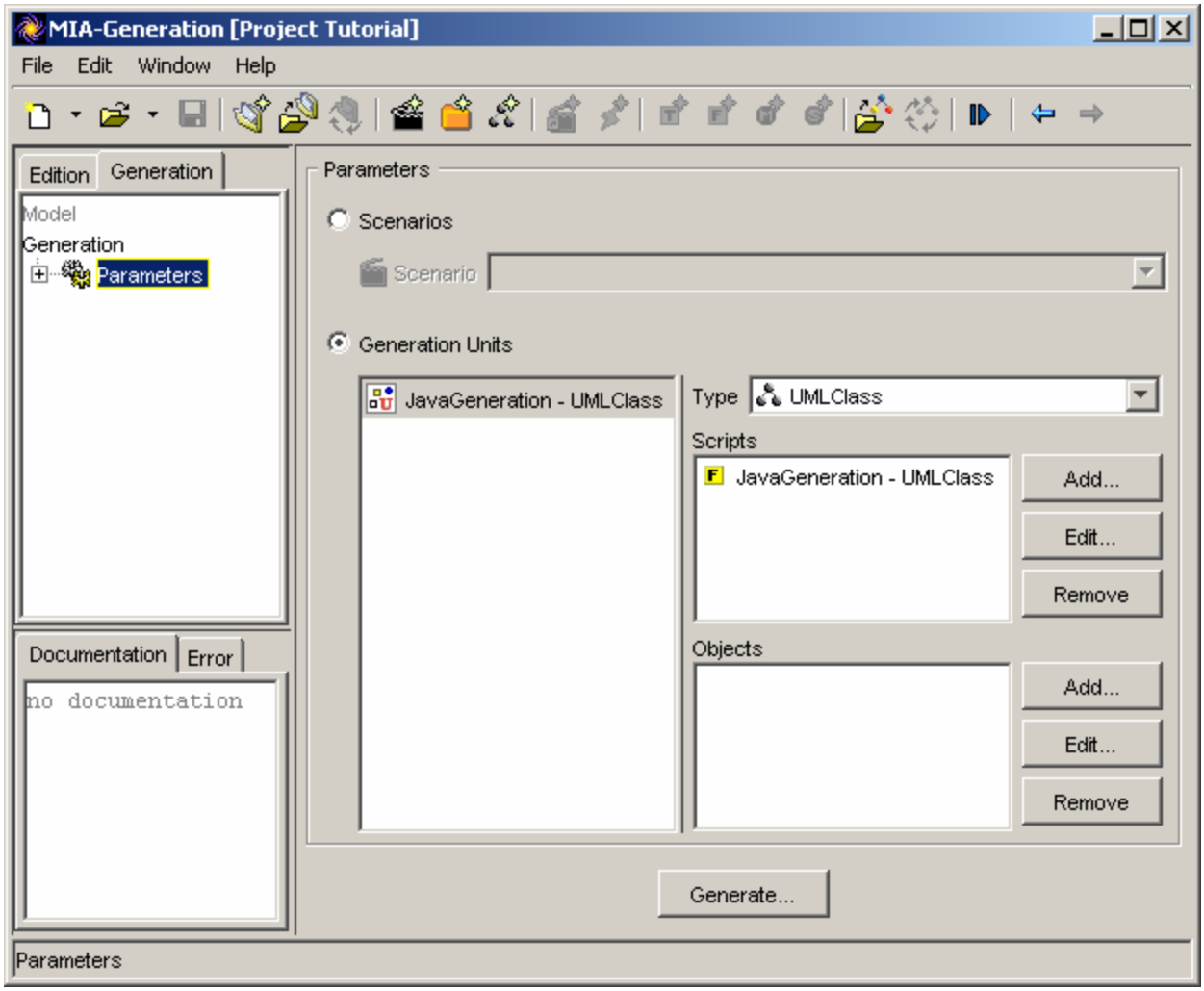


Figure 15 – Editing a Generation Unit

A generation unit has been created for the *JavaGeneration* file template

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Objects Selection

To select objects :

1. Select the type UMLClass from the **Type** combo-box : you will test scripts defined for this type.
2. Click the **Add…** button from the **Objects** list :

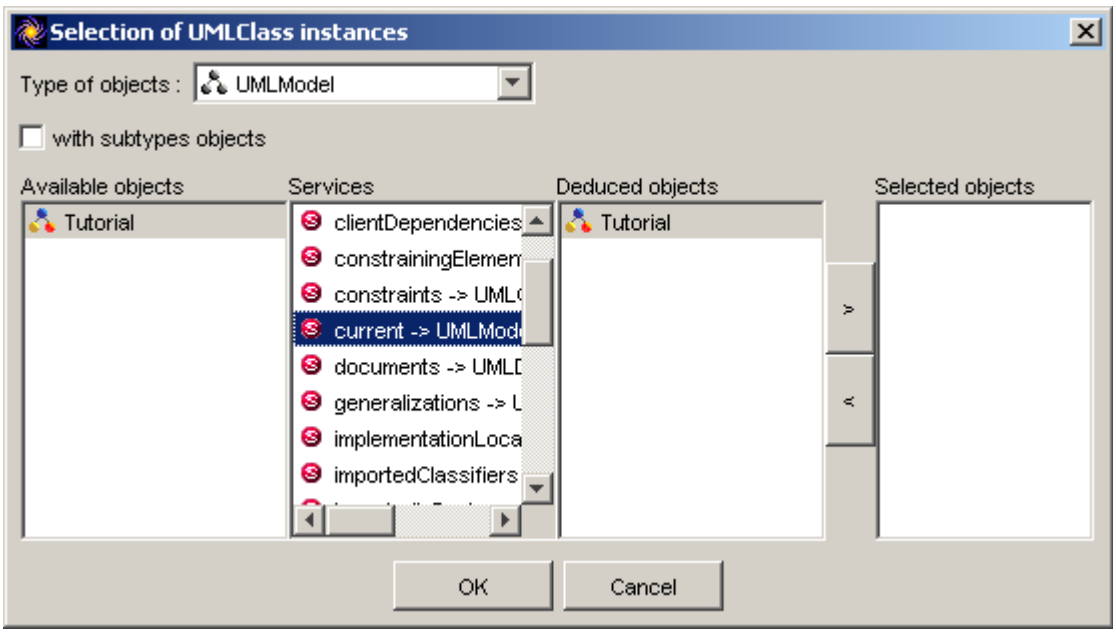


Figure 16 – Objects Selection

You can see required objects type in the dialog title. We want to test a script defined on UMLClass type : we must select UMLClass instances.

In the **Type of objects** combo-box, choose the type of objects that you wish to select. All objects of the selected type will then appear in the **Available objects** list. If you also wish to view the objects of the subtypes, select the **with subtypes objects** checkbox.

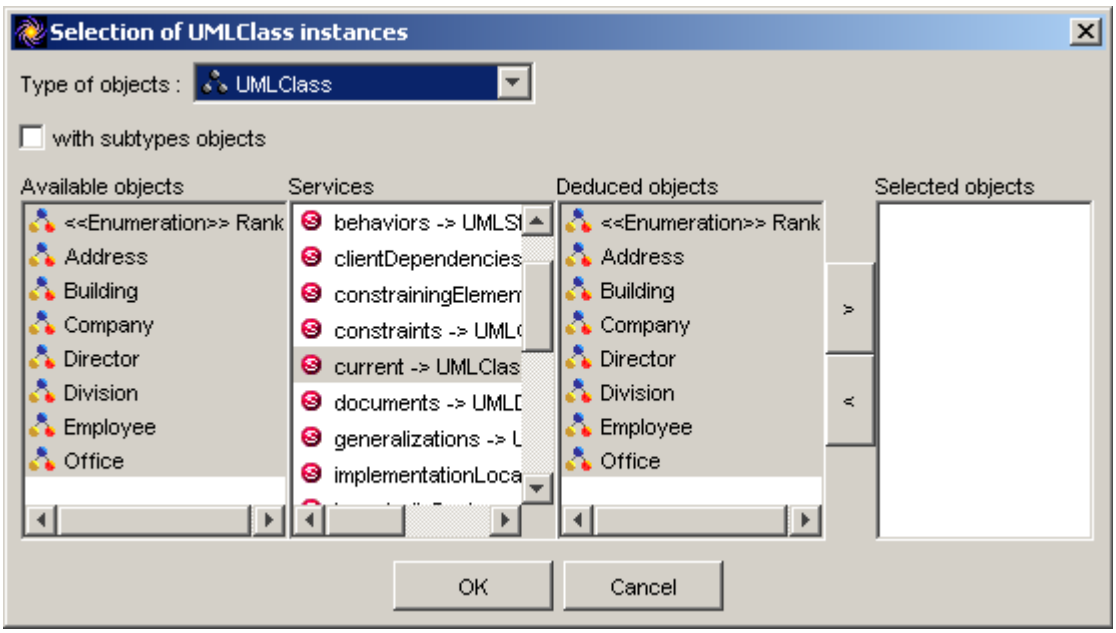


Figure 17 – UMLClass objects

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Then select a service from the ones available for the selected type. Applied to the objects selected in the **Available objects** list, the service may allow you to derive other objects. By default, the **current** service (available in the standard version) is selected, which makes it possible to deduce exactly the same objects as those selected in **Available objects**.

This mechanism is very useful when you wish to launch a generation on a precise subset of objects. Rather than selecting them by hand, you can write a service that returns them automatically. In this way, for example, you can easily (and without any risk of omission) launch a generation on all persistent classes of a model, or on all subclasses of a particular class. The selected service is automatically applied to each object selected in **Available objects**, and the derived objects are placed in the **Deducted objects** list. (If the same object is derived several times, it will appear only once in this list).

Among these derived objects, you can select exactly the objects on which you wish to launch a generation. To do this, use the arrows, which enable you to add "Address" et "Building" objects.

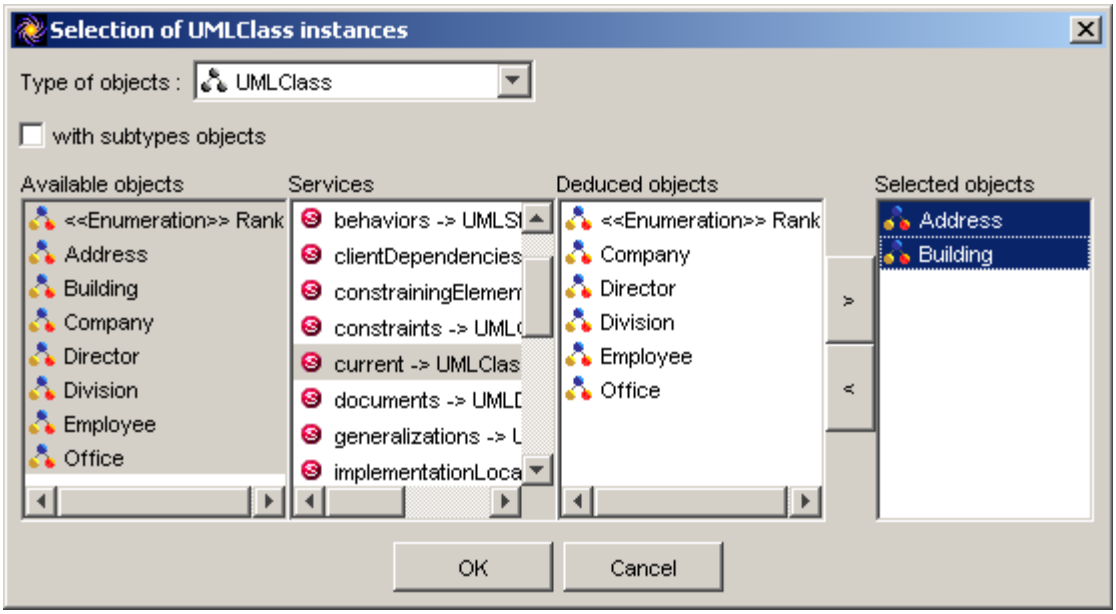


Figure 18 - Address and Building selection

Once your choice is made, click the **OK** button to update the list of the objects in the generation window. (The selected objects will then replace the previous ones).

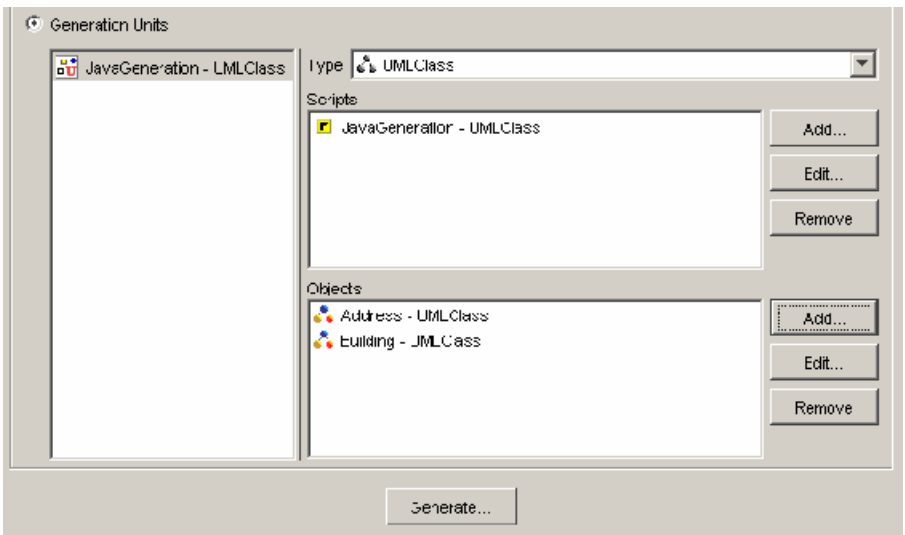


Figure 19 – Selected objects

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Generating a file template

Once the objects and scripts have been selected, you have only to launch the generation using the **Generate** button or the shorcut from the toolbar. Before carrying out a generation, you must choose generation options :

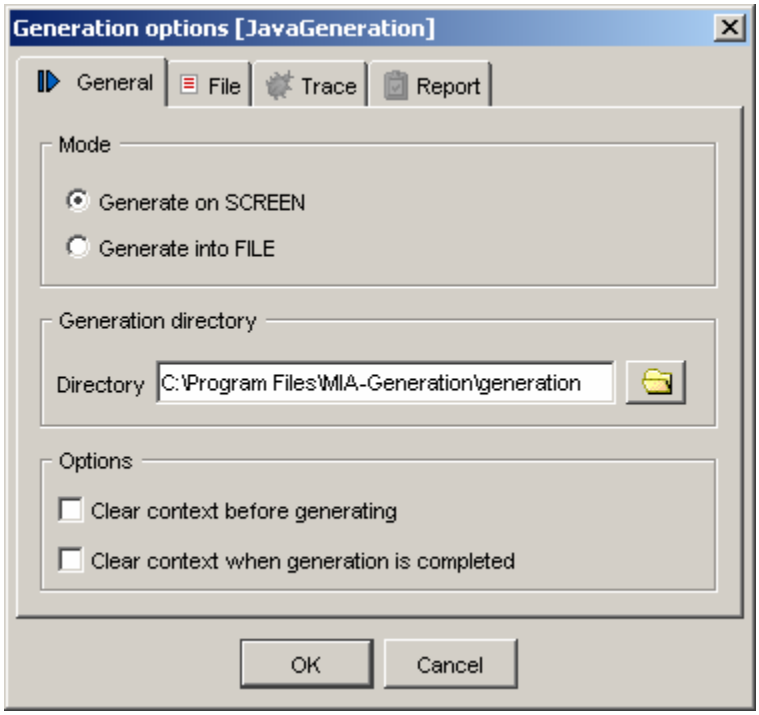


Figure 20 - Selection of the generation mode

Select "Generate on SCREEN" and click on the button "OK".

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The virtually generated filenames appear in the Files list. By selecting a filename, the generated contents will appear in the **Generated text** pane. You can see now that the generated text does indeed depend on the class where the File Template was evaluated.

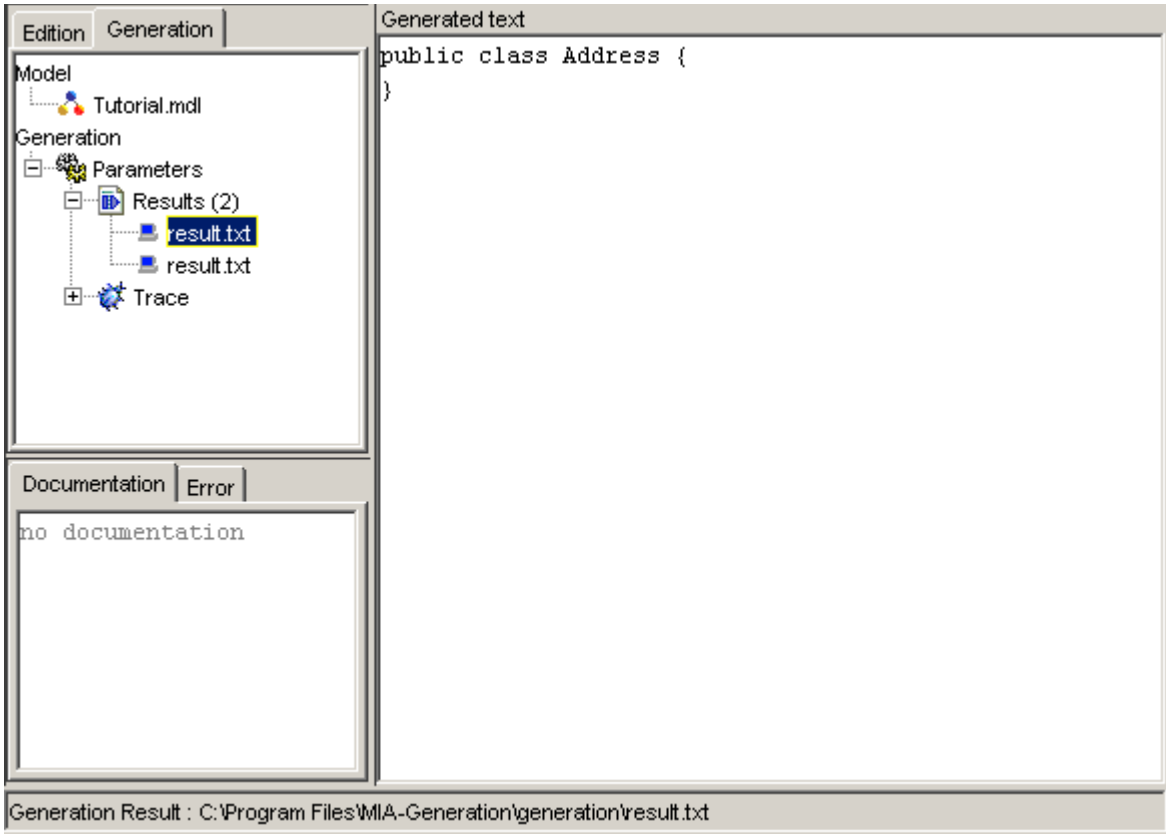


Figure 21 - Results of the generation

All the classes are therefore generated in the same file named "`result.txt`" ! We are going to see how to change the filename property of a file template.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Filename property

In the previous example, the text generated is automatically put into a file named "result.txt". All the classes are therefore generated in the same file !

We are going to change the script Filename of the File Template, a property that determines the name of the file where we want to generate the text.

This script Filename is itself a template. Therefore, we can make a call to the macro "name" so the filename depends on the name of the class.

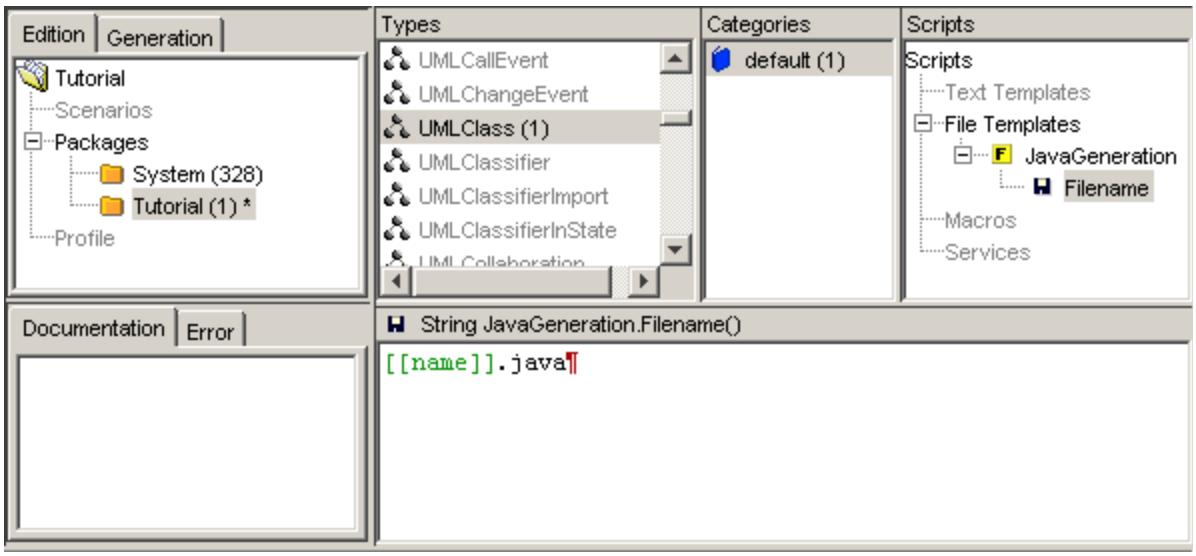


Figure 22 – Filename property

After saving *JavaGeneration.Filename* template, click the Launch Generation button from the toolbar. We produce the correct filenames, taking into account the name of each class :

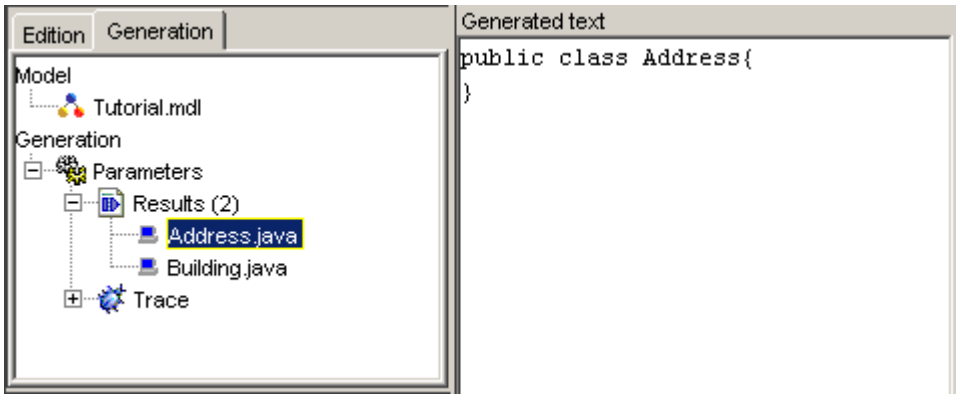


Figure 23 - Results of the generation with the correct filenames

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Improving the Java Generation

Inheritance Generation

We are now going to improve our Java generation to include class inheritance. We must insert the name of the superclass in the generated text for the current class, or "Object" if there is no superclass.

Creation of a superClasses service

To generate name of the superclass in the generated text, we have to get parents from a class in the loaded model. **Services** are special macros that enables you to return objects from your model.

Create a service, named "superClasses", on the type *UMLClass*, using the button . A service is a script written with in Java™, that does not return a string, but one or more objects from the model. You must specify the type of return object for the service (select *UMLClass*) :

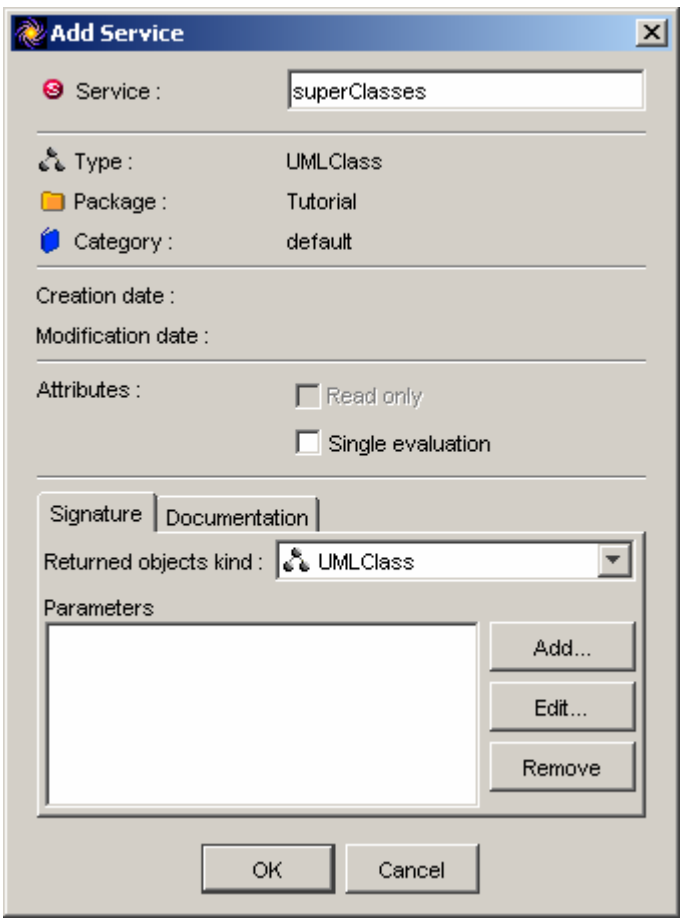


Figure 24 – Service properties

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

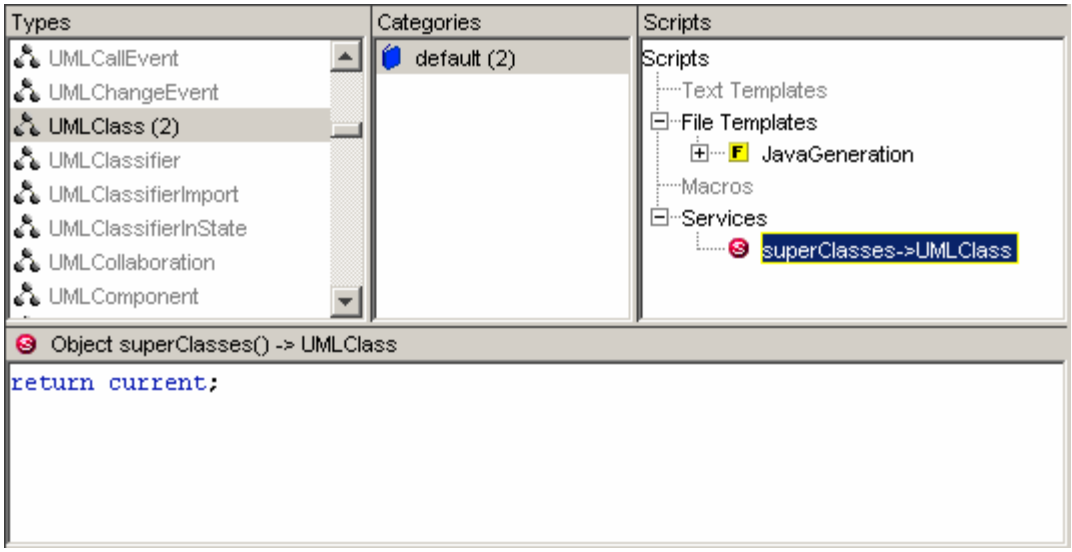


Figure 25 - Création d'un service

This service must return the superclasses of the current class.

In order to see how an inheritance link is instantiated in UML, go to the Model browser. We are going to look at Employee relationships.

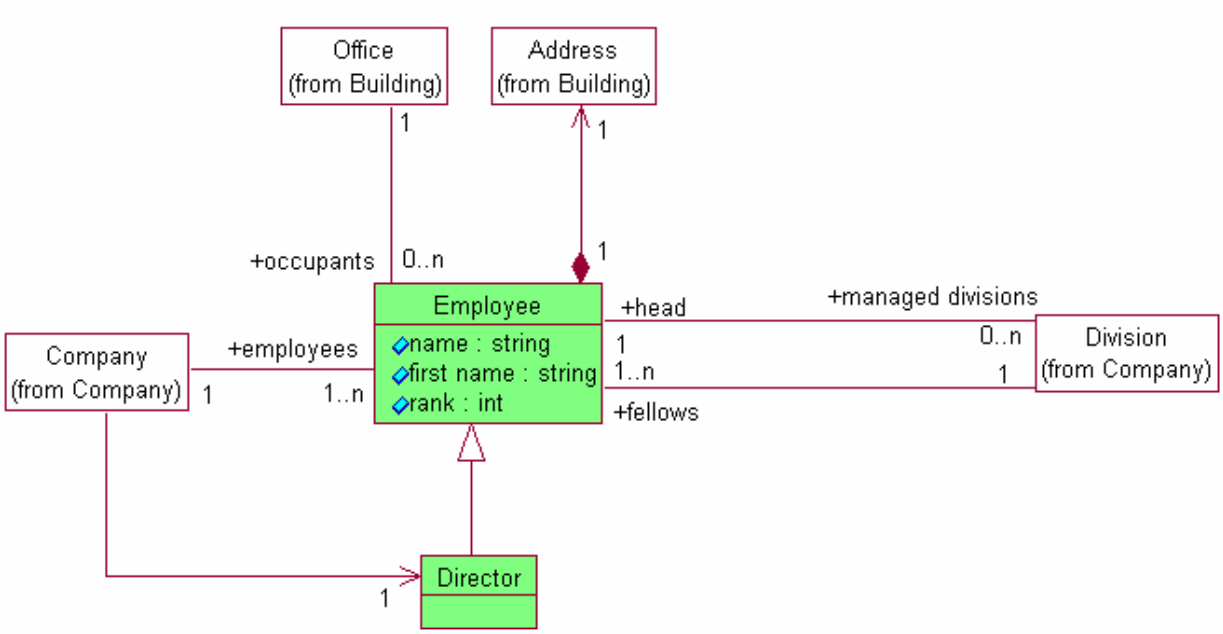


Figure 26 – Employee relationships

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

If you select the object "Director" of type "UMLClass", you can see a link "generalizations", which relates this class to an object *UMLGeneralization*.

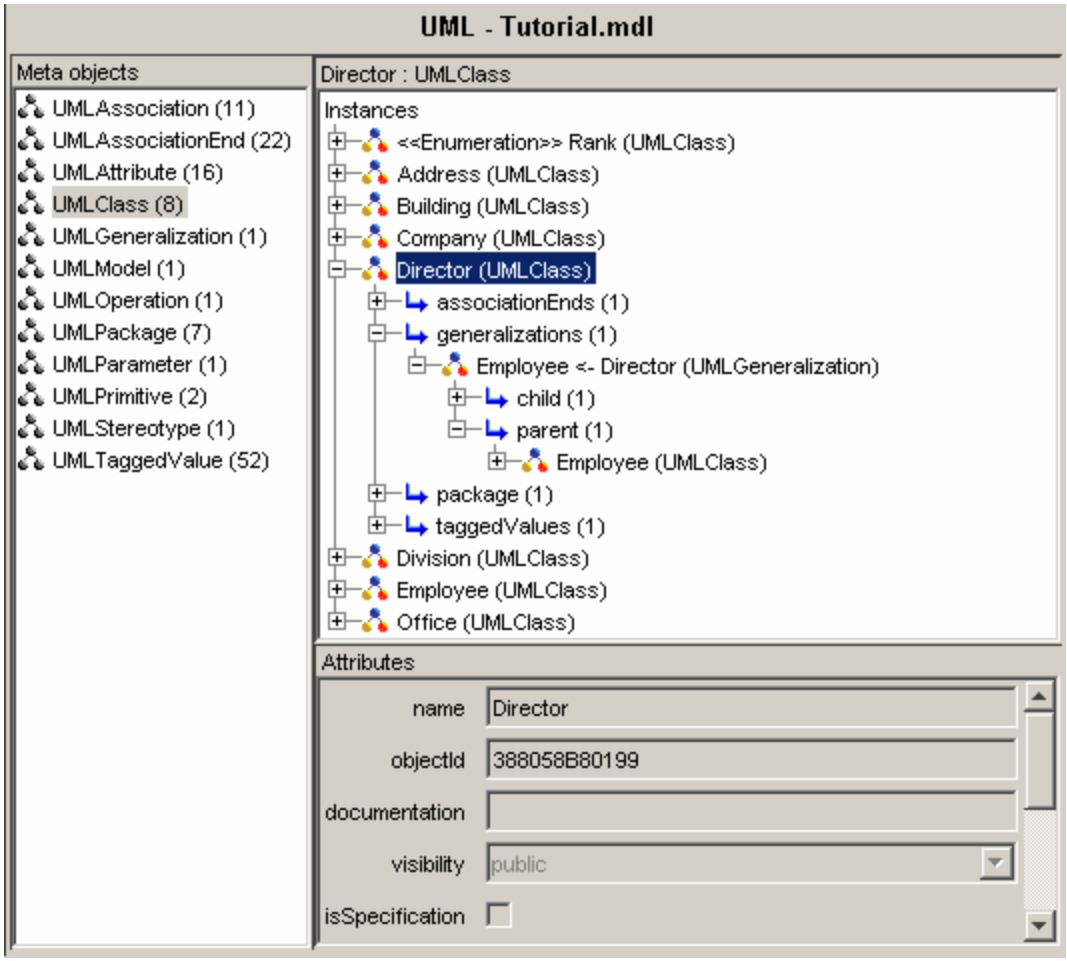


Figure 27 – UML Generalization

If you go into this object, you will see a link "child" to the subclass and a link "parent" to the superclass.

This links *generalizations*, *child*, *parent* are those defined in the UML metamodel :

Figure 28 – "Generalization" links

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

You can find this diagram ant others using the menu "Help > Metamodel APIs".

You can display it by selecting it from the **Window > MetaModel Browser** menu.

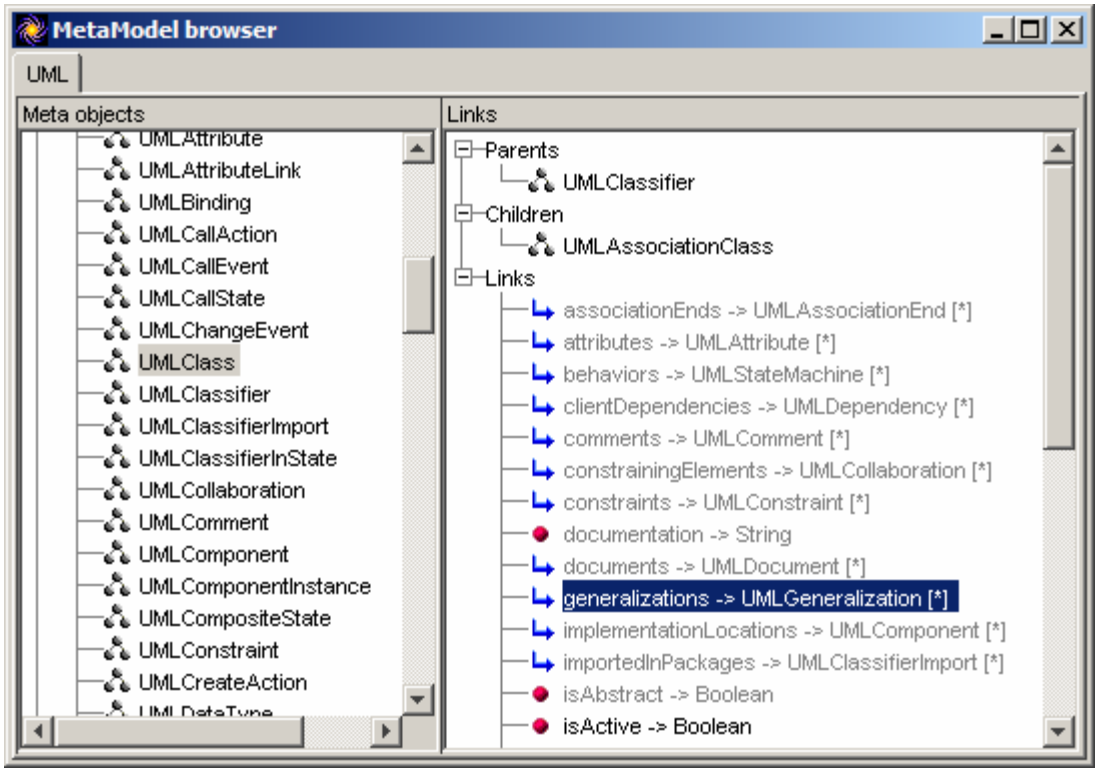


Figure 29 - Browser de méta-modèle

Metamodel and "System" scripts

The browser is compound of two areas : the list of the metamodel types (on the left) and the available links and attributes for the selected type (on the right).

For any type of model object, a certain number of basic accessors are provided. For every link (↳) in the metamodel, there is a service in the "System" package returning the object(s) linked to the role (under an object table format if there are many objects). For each attribute (●) in the metamodel, there is a macro returning a String that represents the attribute value.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The service must therefore go through the list of objects attached to the current class by the link "generalizations", and return the object "parent" of each generalization.

Object superClasses() -> UMLClass

```
List generalizations = current.getList("generalizations");

List parents = new ArrayList();
for (Iterator i = generalizations.iterator(); i.hasNext(); ) {
    UMLGeneralization generalization = (UMLGeneralization) i.next();
    parents.add(generalization.getParent());
}
return parents;
```

Figure 30 - Body of the superClasses service

We call the service "generalizations" (defined in the “System” package) by using the method ***getList***, which returns an instance of *java.util.List*.

The **getList** method will format the returned objects from a service into an instance of *java.util.List*. If the service only returns one object, **getList** returns a list containing this object. If the service returns an iterator or array, they will be converted to a *List.* Using this method, it is possible to call all services in the same manner without having to know how they were implemented.

Difference between returned type of a service and returned objects kind

The declared **Return Type** is **java.lang.Object** but the real service return type can be of several kinds :

- a model object (subtype of MIAObject),
- an object array (Java type = Object[]),
- a collection of objects (Java type = *java.util.Collection*),
- a series of objects (Java type = *java.util.Iterator*).

MIA-Generation does not automatically determine the kind of the returned objects. So, at the time of the creation of a service, it is necessary to define which kind of model object it returns. If the service returns a collection of *UMLClass* instances, the **Returned Objects Kind** will be *UMLClass* (even if the returned object is an instance of *Collection*).

MIA-Generation provides a method ***collect*** to make it easier to write this type of service. This method *collect* permits you to call a script on each object returned by a service, and to retain the results from each execution of the script. So a call to *collect("generalizations", "parent")* returns the list of parents from each generalization of the current class.

Object superClasses() -> UMLClass

```
return current.collect("generalizations", "parent");
```

Figure 31 - Service using the method collect

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating a superClassName macro

We now have to write a script calling this service, which will return the name of the first superclass or "Object" if no superclass exists.

Therefore we are going to create a macro, which will allow us to write some Java code that returns a string. This can then be inserted into a template.

Create a macro, named "superClassName" on the type *UMLClass*, using the button from the toolbar. Specify the return type *String* for the macro.

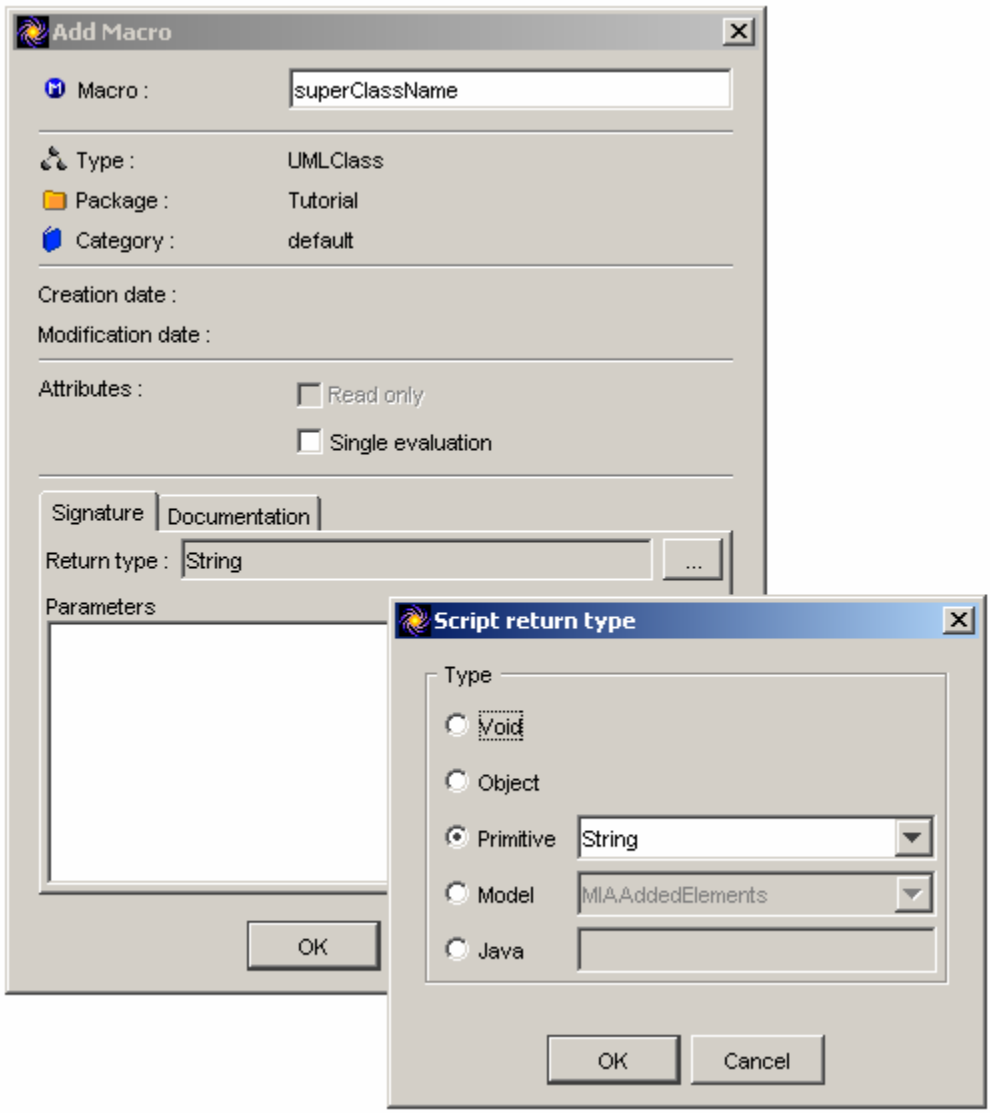


Figure 32 - Creation of a superClassName macro

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

This macro calls the script "name" (defined in the package "System") using the method **getString**, which returns the name of the class. With the method getString, it is possible to call a template or macro and to get the string that results from its evaluation.

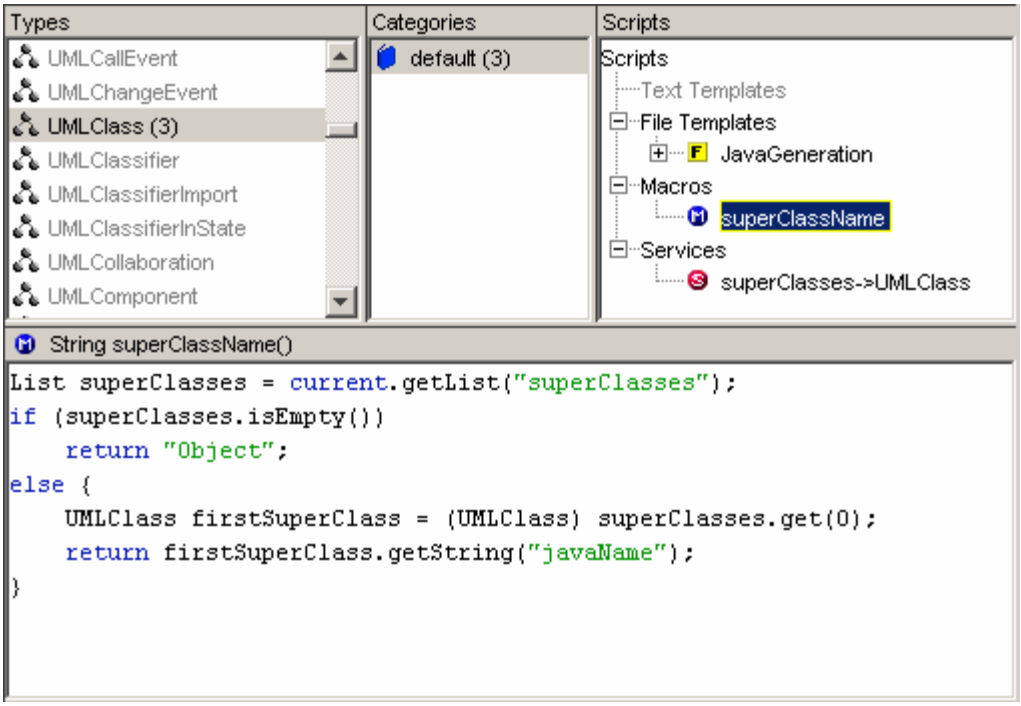


Figure 33 - Creation of a macro superClassName

Call the macro superClassName

This macro still needs to be called from the “Java Generation” File Template.

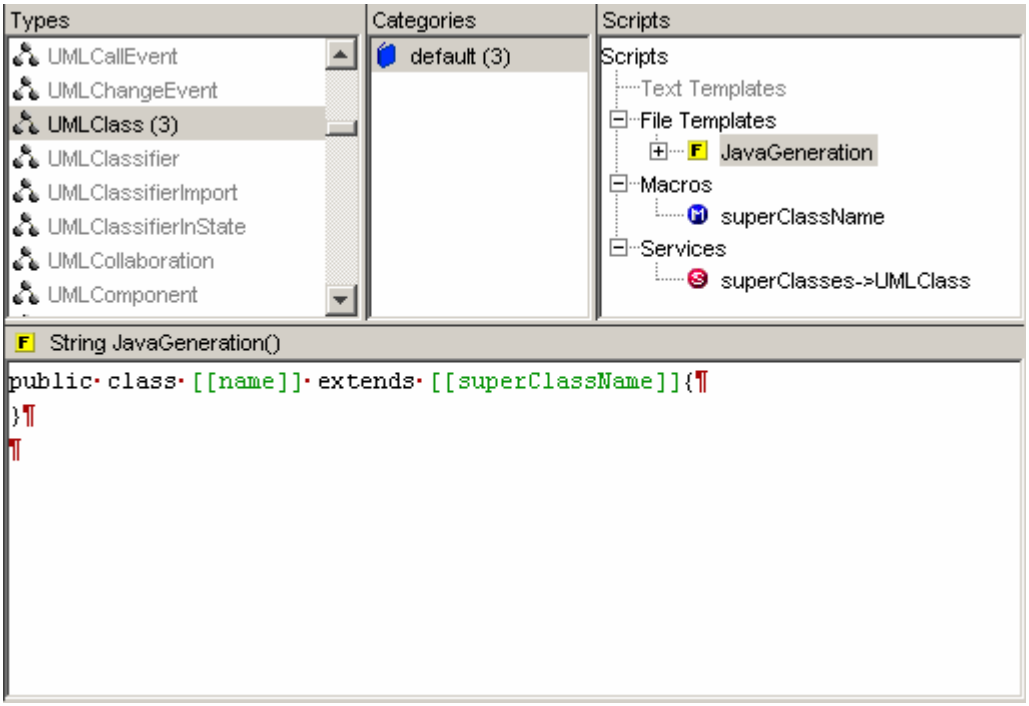


Figure 34 - Call to the superClassName macro

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Generating trace

If we now regenerate the files with the classes "Address" and "Director", the Java class "Address" extends the "Object" class, and the Java class "Director" extends the class "Employee". You can activate trace at the time of generation.

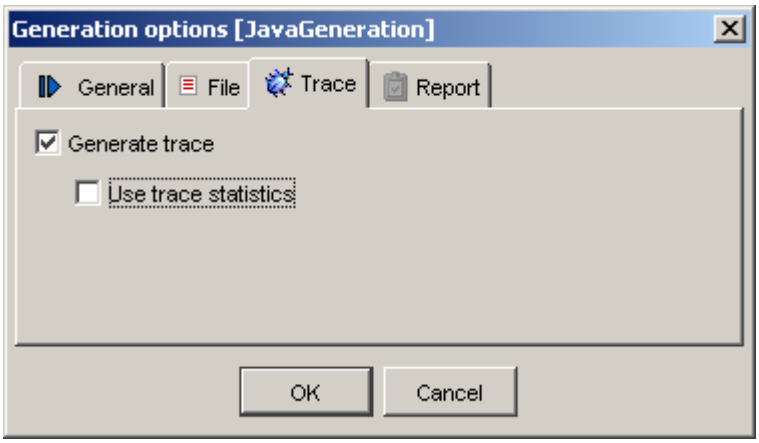


Figure 35 – Traces options

If we now regenerate the files with the classes "Address" and "Director", the Java class "Address" extends the "Object" class, and the Java class "Director" extends the class "Employee".

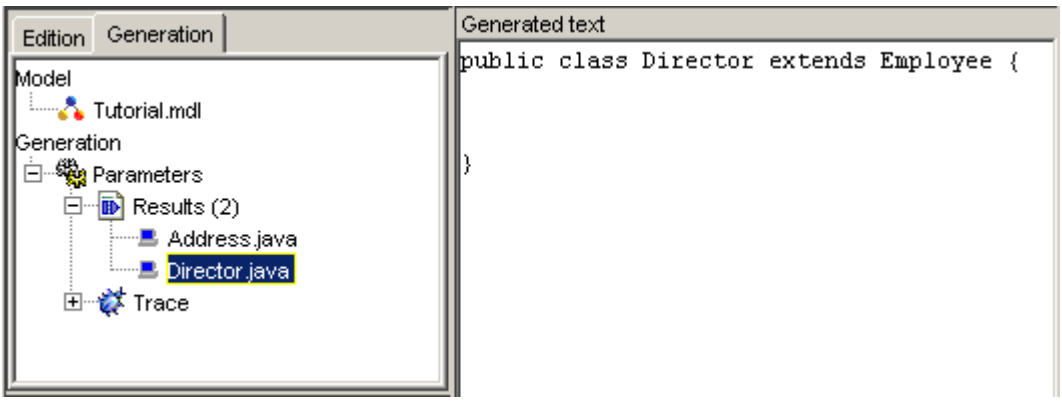


Figure 36 - Generation of the superclass

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

A trace contains a list of generation nodes. Each node describes the call of a script on an object of the model. If the generation has been traced (*Generate trace* option was enabled at the time of generation), you can browse the trace in the Navigator tree from the Generation tab.

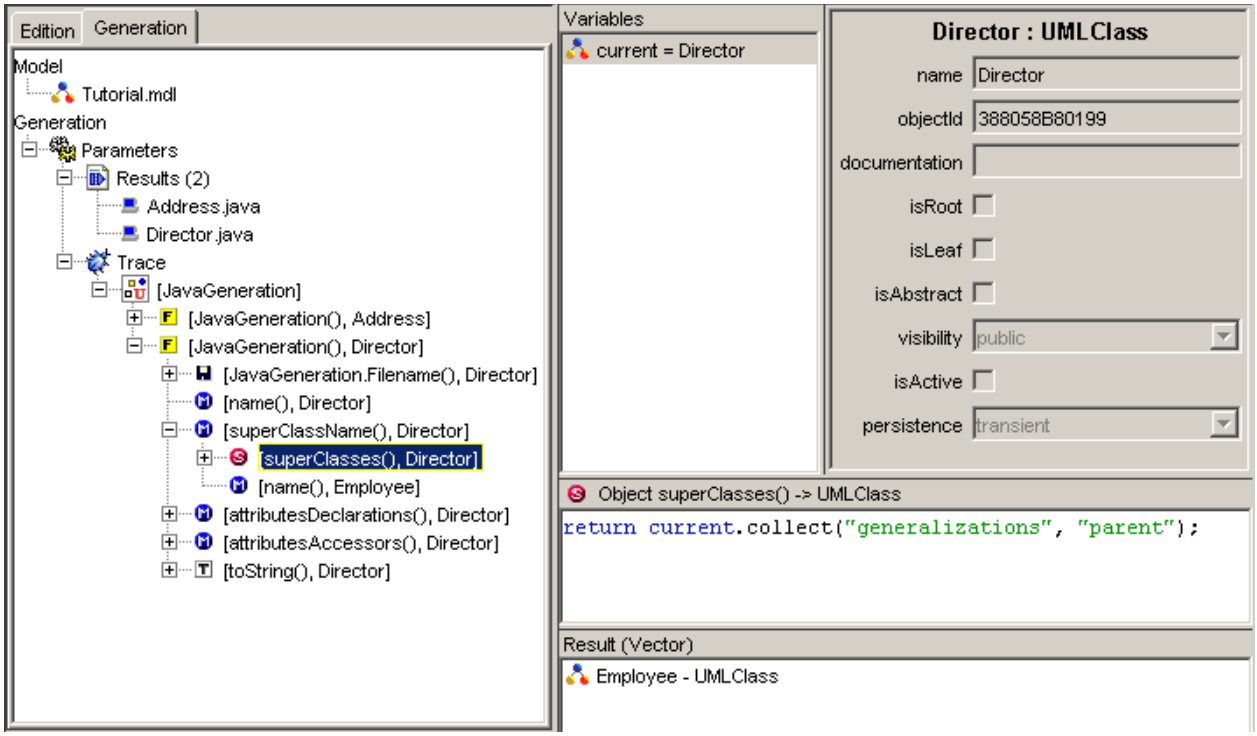


Figure 37 - Generation trace of JavaGeneration

Each nodes contains :

- The object of the model on which the script has been called.
- The called script.
- The text generated by the call of the script on the object.
- The nodes created by the call (the call can launch other scripts calls).

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Attribute Generation

We are going to improve our Java generation to add the declaration of attributes and some attribute accessors.

Adding a template for attribute declaration

1. In the **Types** list, select the type *UMLAttribute.*
2. Create a template called "declaration" using the New Template button .
3. Change the generated text of the template so that it declares the visibility of the attribute, followed by the name of this attribute. Insert a carriage return at the end of the template so that each attribute is declared on a new line.

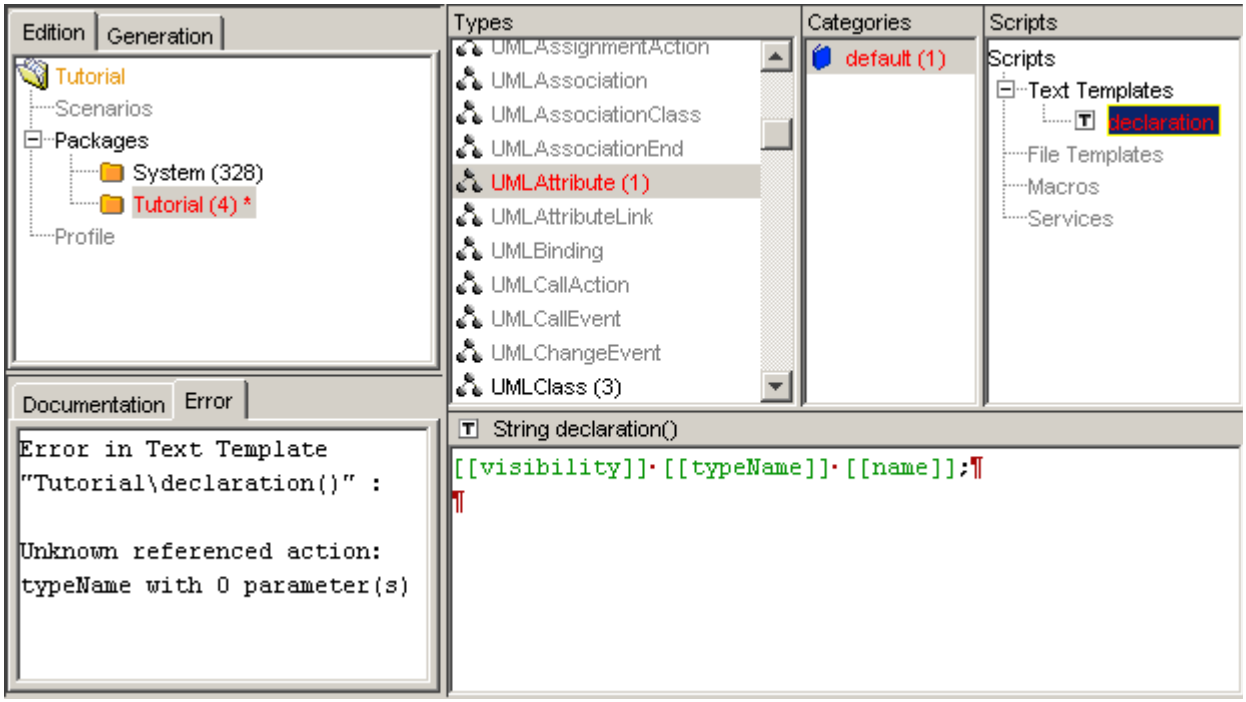


Figure 38 - Template for an attribute declaration

Note : The *visibility* macro is defined in the "System" package.

Be aware that MIA-Generation will notify you of an error after creation of the script “declaration” because the scripts “typeName” does not exist. After creating this script, the error will be removed.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The macro *typeName* returns the attribute type name using the link named "type" between an attribute and its owner.

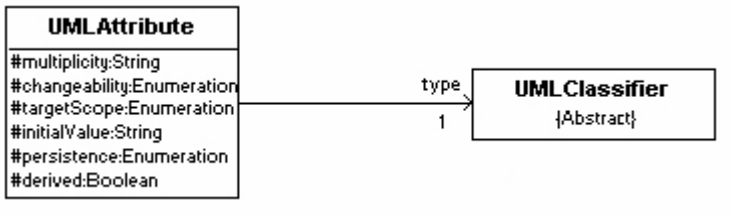


Figure 39 - "type" link

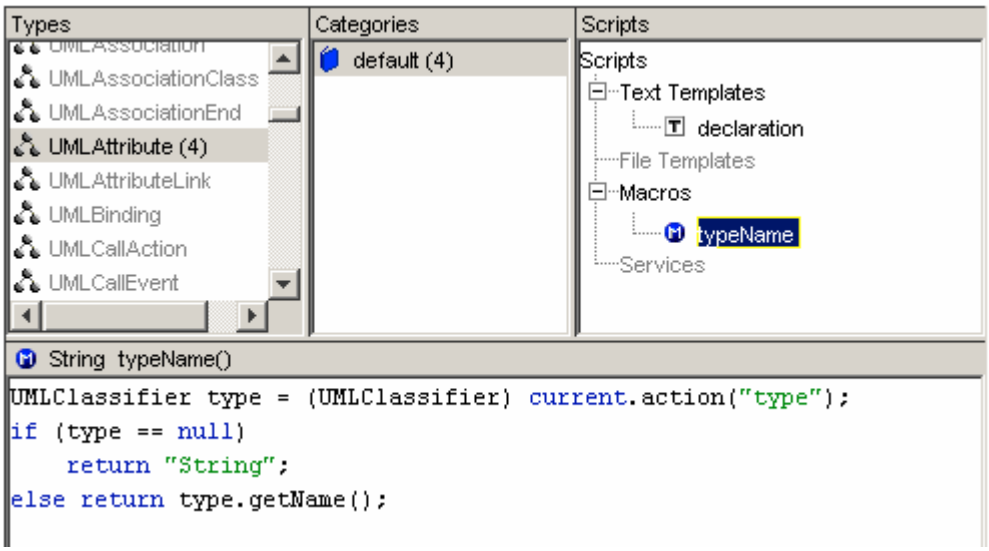


Figure 40 - Macro typeName

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Adding a template to generate the accessors

Create a second template named *accessors* to generate accessors for this attribute :

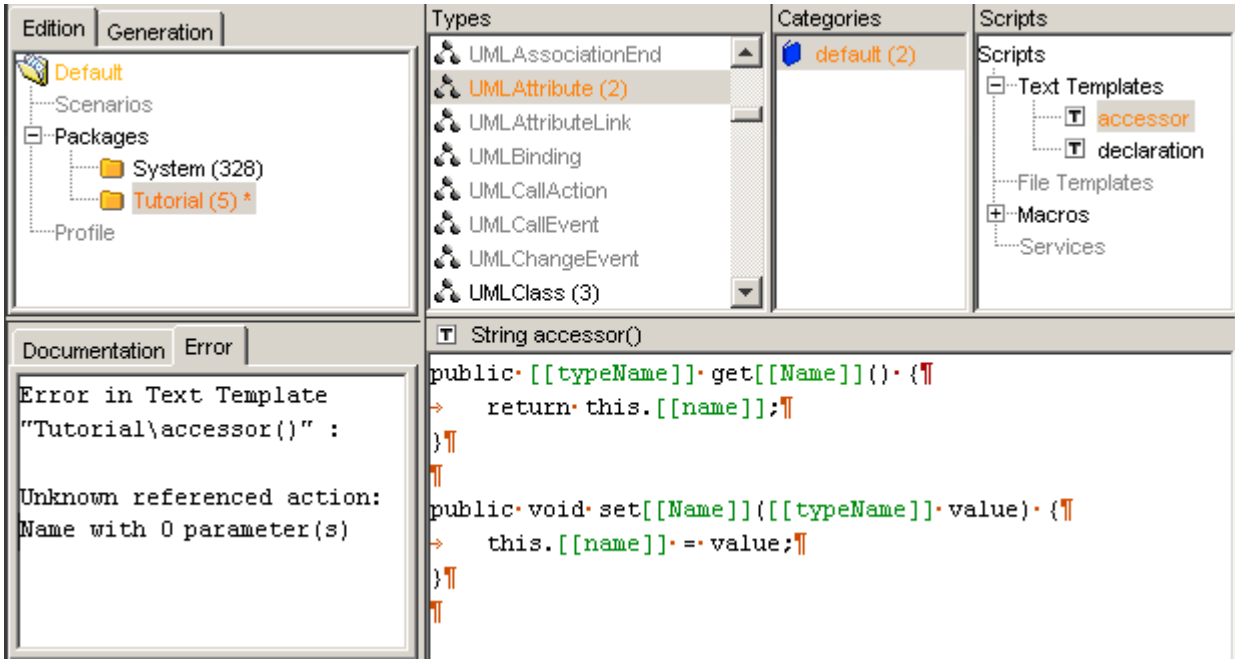


Figure 41 - Template for the declaration of the accessors

This template generates the methods *get* and *set* for an attribute. The script uses a macro "Name", putting the first letter of the attribute name as a capital letter, and a macro "typeName", which returns the attribute type name.

The *Name* script puts the first letter of the attribute name as a capital letter.

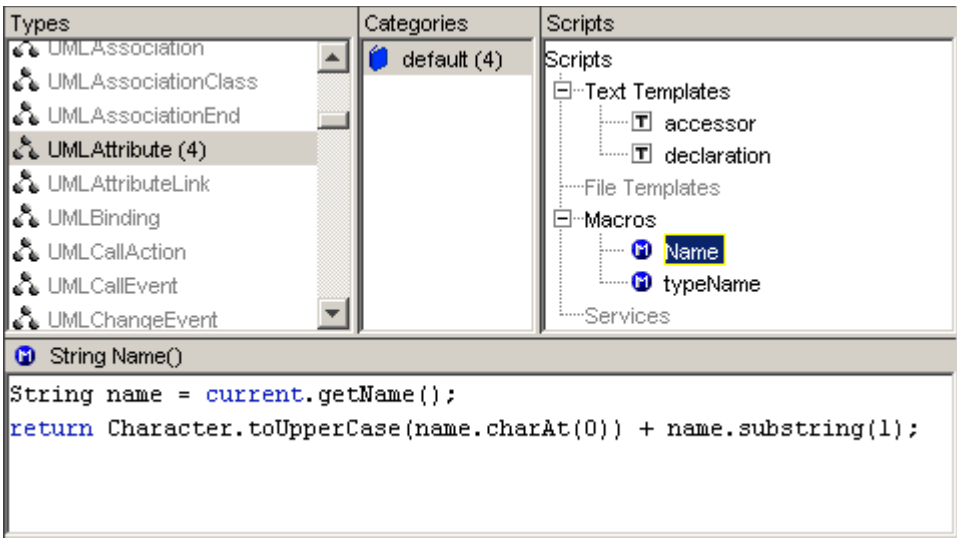


Figure 42 - Macro Name

This "Name" macro could have been defined on the type *UMLModelElement*, parent of all UML types, where the method "name" is defined, because it's a function that is very general and can be used for attributes, classes, associations, etc…

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Generation of the attributes from a class

1. Add a macro "attributesDeclarations" on the type *UMLClass*.

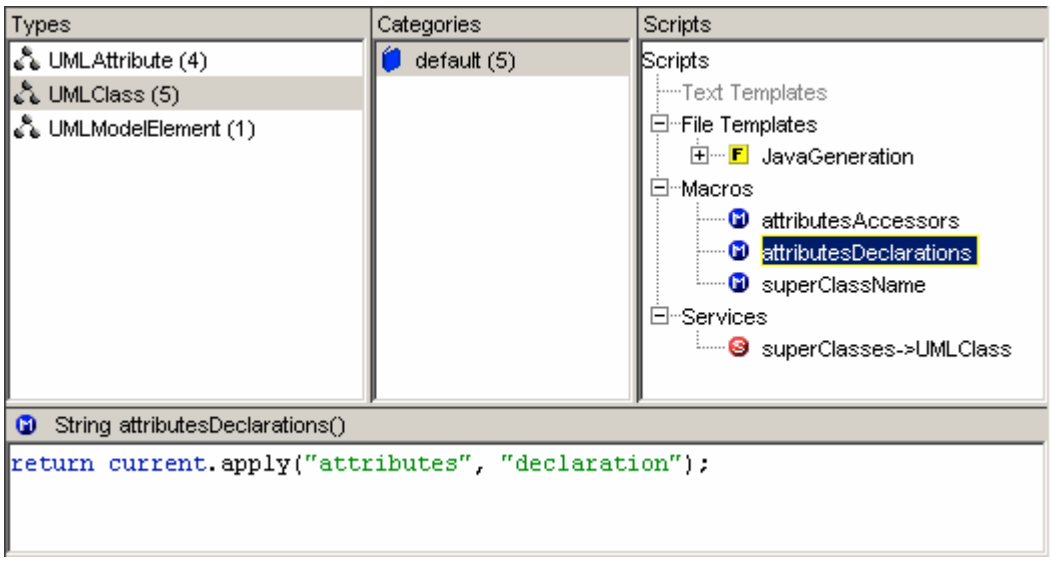


Figure 43 – attributesDeclarations macro

This macro uses the method ***apply*** to call the template "declaration" in each *UMLAttribute* object returned by the service "attributes" (defined in the package System), and concatenates the results in the form of a string. The result is then returned.

Note :

It is possible to mask the types for which no script is defined (context menu **View > Mask empty types**), excluding System package .

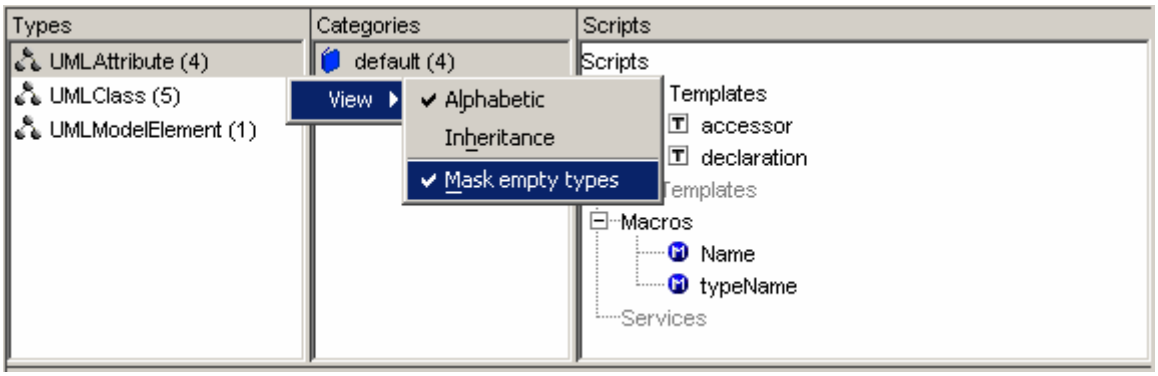


Figure 44 - Display of types defining at least one script

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

2. Add a second macro "attributesAccessors" to allow the generation of the accessors for each attribute of the current class.

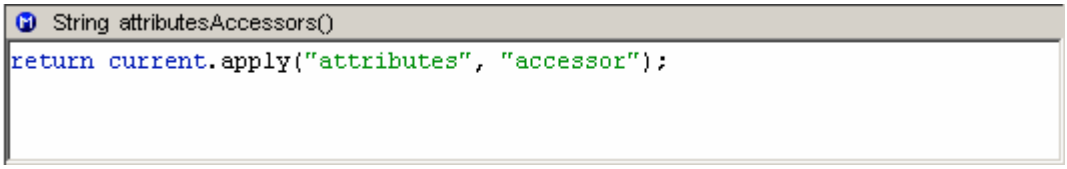


Figure 45 – attributesAccessors macro

3. Modify the File Template "JavaGeneration" so it calls those two macros. A code completion mechanism can help you create scripts. By pressing the "**Control**+**Space**" keys you can display a window containing the scripts you can use in this script. In a template, you can ask for this help after entering the start delimiter of a script (**[[** by default) :

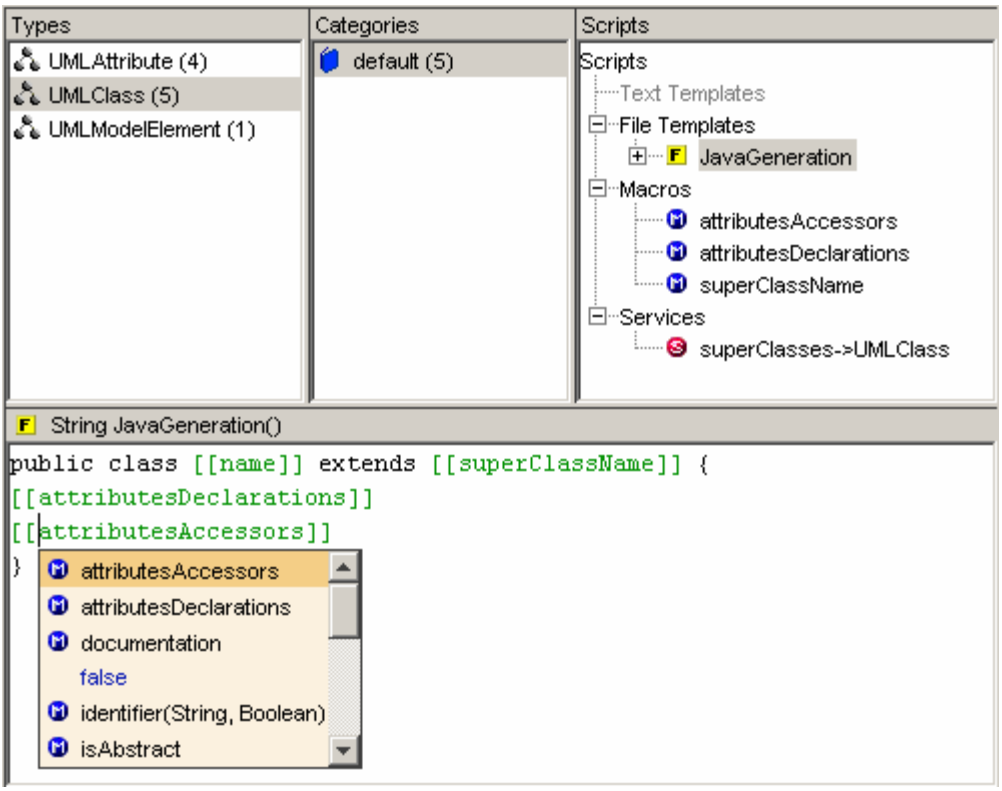


Figure 46 - Include the attributes when generating a class

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

If we generate this File Template on the class "Employee", it creates the following text :

Generated text

```
public class Address extends Object {
public int number;
public String street;
public String zip code;
public String city;
public String country;

public int getNumber {
    return this.number;
}

public void setNumber(int value) {
    this.number = value;
}
public String getStreet {
    return this.street;
}

public void setStreet(String value) {
    this.street = value;
}
public String getZip code {
    return this.zip code;
}
```

Figure 47 - Results of the generation with the Employe class

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating a macro with parameters

We are going to improve our generation by modifiying the *Name* macro to allow us to puts the first letter of the attribute name as a capital letter and replacing some characters in the attribute name.

Create a new macro *String identifier(String id, Boolean firstletterToUppercase)* on the *UMLModelElement* type :

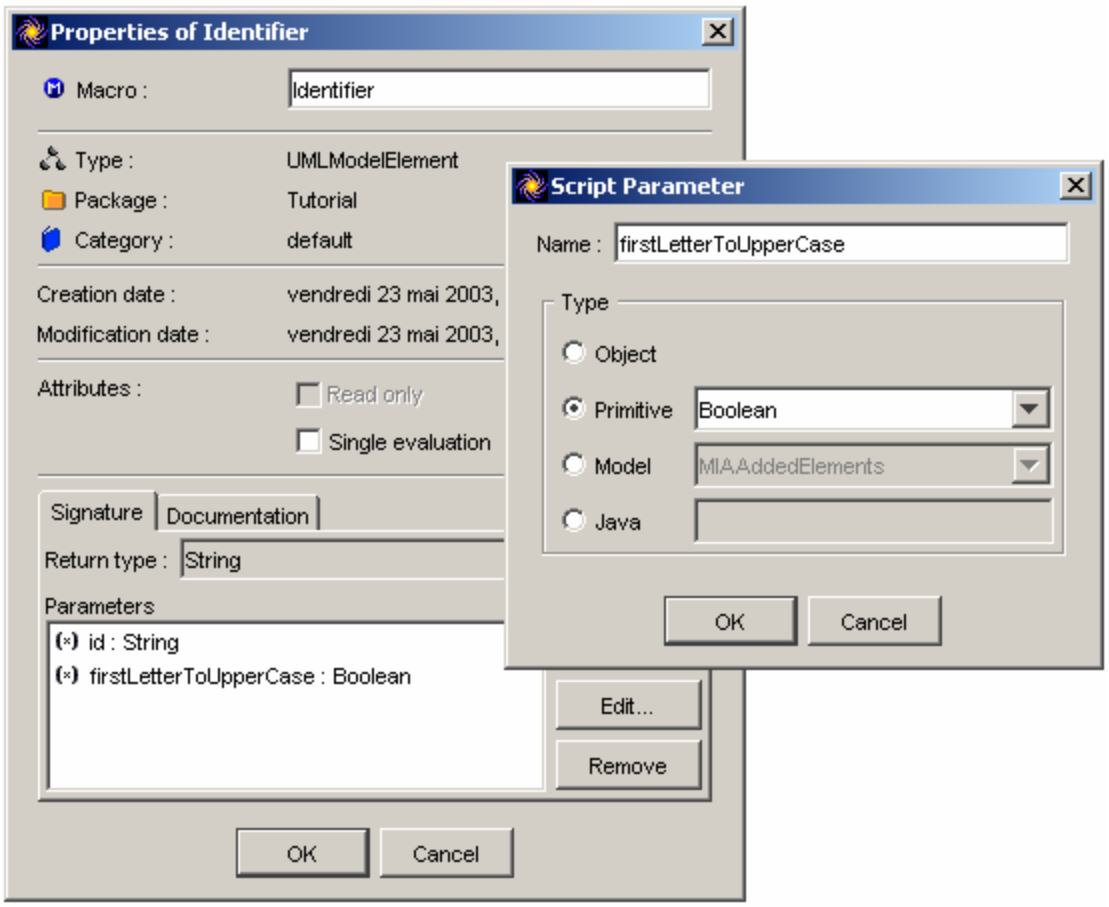


Figure 48 – Creation of a macro with identifiers

Parameters naming

The name of a script or a parameter must be a valid Java™ identifier. current and context keywords are reserved.

Parameters type

Parameters must be Java™ objects (instances of java.lang.Object). To call a script with parameters from a template, MIA-Generation convert primitive types to wrappers (ex : int to Integer, boolean to Boolean).

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The i*dentifier(String,Boolean)* macro is the following :

String identifier(String id, Boolean firstLetterToUpperCase)

```
if (id == null)
    return "";
else if(id.length() == 0)
    return id;

id = id
    .replace('à','a')
    .replace('é','e')
    .replace('è','e')
    .replace('ê','e')
    .replace('ë','e')
    .replace('ï','i')
    .replace('ö','o')
    .replace('ô','o')
    .replace('°','o')
    .replace('û','u')
    .replace('ù','u')
    .replace('ü','u')
    .replace('-','_')
    .replace(' ','_')
    .replace('\'','_')
    .replace('-','_')
    .replace('\'','_');

if (firstLetterToUpperCase.booleanValue())
    id = Character.toUpperCase(id.charAt(0)) + id.substring(1);

return id;
```

Figure 49 - Identifier(String, Boolean) macro

In Java™ scripts, parameters can be used as any other variable.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Calling a script with parameters in Java scripts

There are two ways to reference scripts in constructs that handle scripts calls (action, apply, etc.). You can reference a script using :

- a **String** object : this can be used to reference a script by its name, assuming this script does not expect any parameter.
- a **Call** object : this is to be used to call a script that expects parameters.

Also, to call a script with parameters from another Java™ script, you must use a **Call** object, instance of **mia.gen.openapi.Call**. This class is aimed to create Script calls, to be used in MIAObject contructs (like `action`, `apply`, etc.). Note that creating a Call instance does not directly invoke the referenced Script. The evaluation is done using a MIAObject construct, like the `action` method. Althought a Call instance can refer to a Script that does not expect any parameter (using the `Call(String)` constructor), this class is mainly aimed to call script which expects evaluation parameters.

A typical use may be :

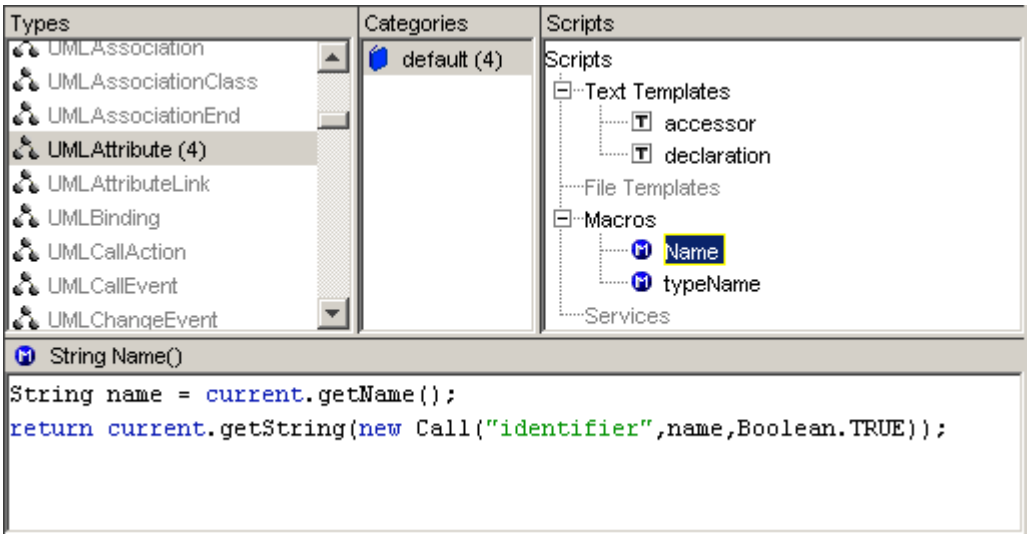


Figure 50 – Calling identifer(String,Boolean) macro from a Jave™ script

Create a script *name* on *UMLAttribute.* You can press **CTRL+C** and **CTRL+V** to get a copy named "*Name1"* from the macro *Name*. Rename it using **Rename** or **Properties** context menus from the Scripts list.

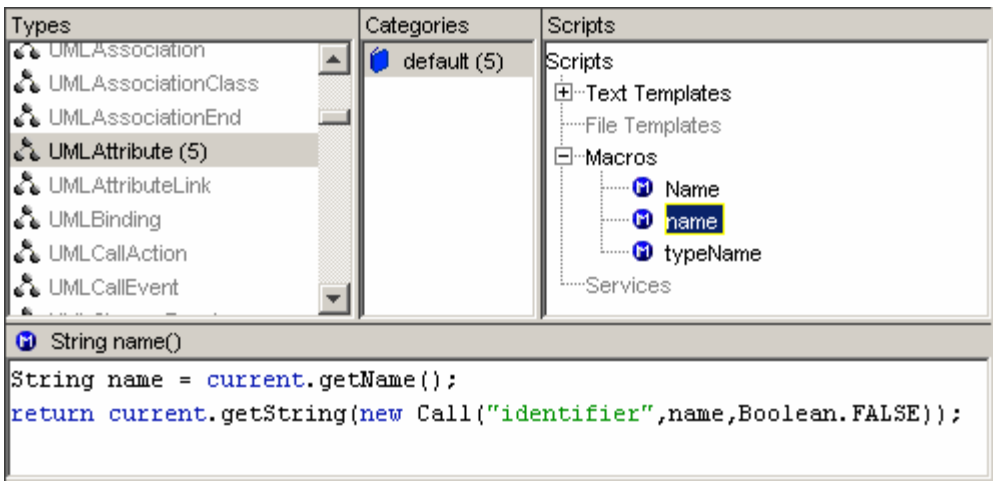


Figure 51 - Macro name

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Calling a script with parameters in a template

To call a script with parameters from a template, you must respect following notation:

- Scripts can only be called on the *current* object.
- To use script with parameters, it must be framed by special characters (**[[** and **]]** by default).
- As a script can be itself a parameter evaluated on the current object of the model, it is necessary to have access to parameters with a special keyword : *args*. You must use the keyword *args* to reference a parameter in a template (ex : `args.myParameter`).
- If you put directly a parameter into a template (ex : `[[args.myParameter]]`), parameter is converted to a String using the *java.lang.String.valueOf(Object)* method.
- Parameters must be called between brackets (ex : `[[myScript(args.myFirstParameter, args.mySecondParameter)]]`).
- A script without parameters can be called with or without brackets (ex : `[[myScript]]` or `[[myScript()]]`).

Primitive types and String used as parameters

You can use primitive types in a template (int, boolean, etc.) or strings as parameters. They will be convert into Java™ objects (because a parameter must be an instance of Object) :

Expression	Java Object
true	`java.lang.Boolean.TRUE`
false	`java.lang.Boolean.FALSE`
1	`java.lang.Integer`
1.2	`java.lang.Float`
'a'	`java.lang.Character`
"text"	`Java.lang.String`
null	`null`

For example, we are going to use a call to *Name* macro for the getter and a call to *identifer(String,Boolean)* inside the template for the setter.

String accessor()

```
public [[typeName]] get[[Name]](){
    return this.[[name]];
}

public void set[[identifier(name,true)]]([[typeName]] value) {
    this.[[name]] = value;
}
```

Figure 52 – Calling identifier(String,Boolean) from a template

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

User Code Management

In general, it is better to architect the generation so that it is not necessary to insert user code in the generated files. But it is impossible in some cases : for example, if it will take more time to model and generate business rules than to code them directly.

MIA-Generation gives the possibility to delimit in the resulting text generation, a zone that will be inserted manually. We delimit this zone by using the scripts BeginTag and EndTag.

Take for example the method *toString()*, which in Java obtains a string representation of an object. It is not possible to determine at generation time what the implementation of this *toString()* method will be. We can however generate a default implementation of this method, and allow it to be modify manually by the following :

Create a template "toString" on the type *UMLClass*.

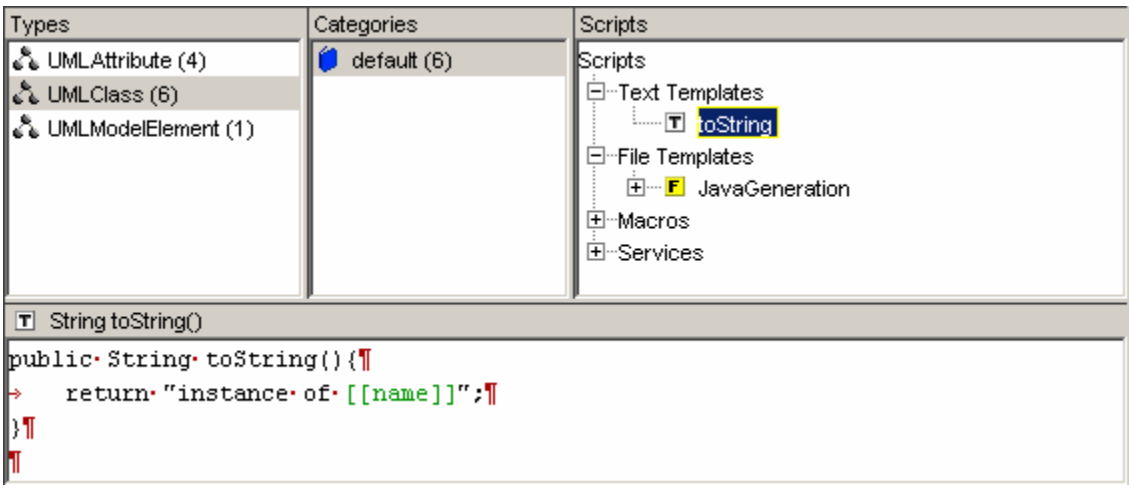


Figure 53 – toString Template

Call this template in the File Template "JavaGeneration".

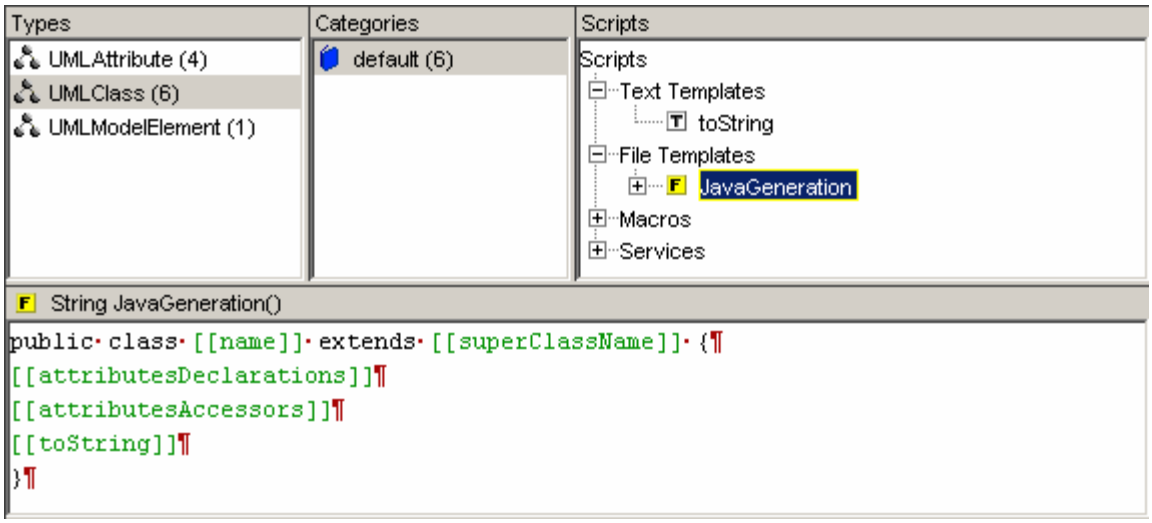


Figure 54 - Appel de toString dans JavaGeneration

At this point, the implementation of the method *toString()* generated is fixed, and all manual modifications will be lost during the next generation.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Use the contextual menu "Add tags" on the "toString" template to add the BeginTag and EndTag scripts.

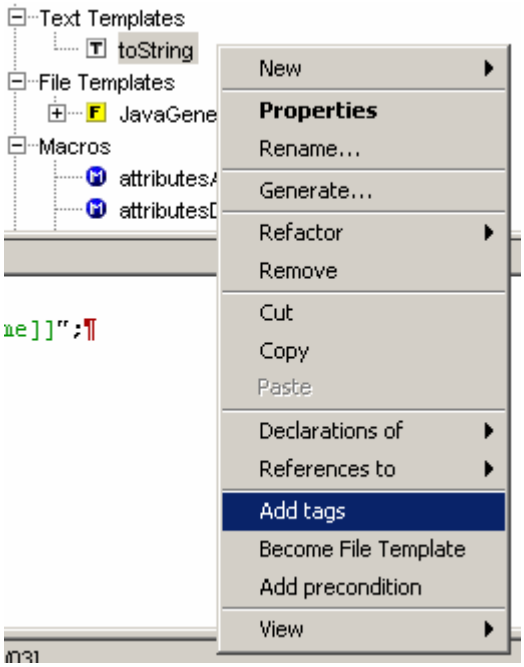


Figure 55 - Adding tags

The scripts **BeginTag** and **EndTag** are templates themselves. We can therefore call other templates or macros in these scripts. The BeginTag script will allow you to delimit the beginning of the text zone that will be manually inserted, and the EndTag will mark the end of this zone.

During generation, if the file does not exist, the code is generated with the tags, and between the tags, the text that is defined in the script between the calls to BeginTag and to EndTag. On the other hand, if the file exists, the BeginTag script is evaluated and if the tag is present in the file, the text which is between the two tags of the file, is recovered and replaced between the two generated tags.

The result of the evaluation of a BeginTag and of an EndTag must be unique in a generation (you cannot, for example, generate a tag "//Start of user code" two times). In addition, **BeginTag and EndTag should not contain carriage returns**.

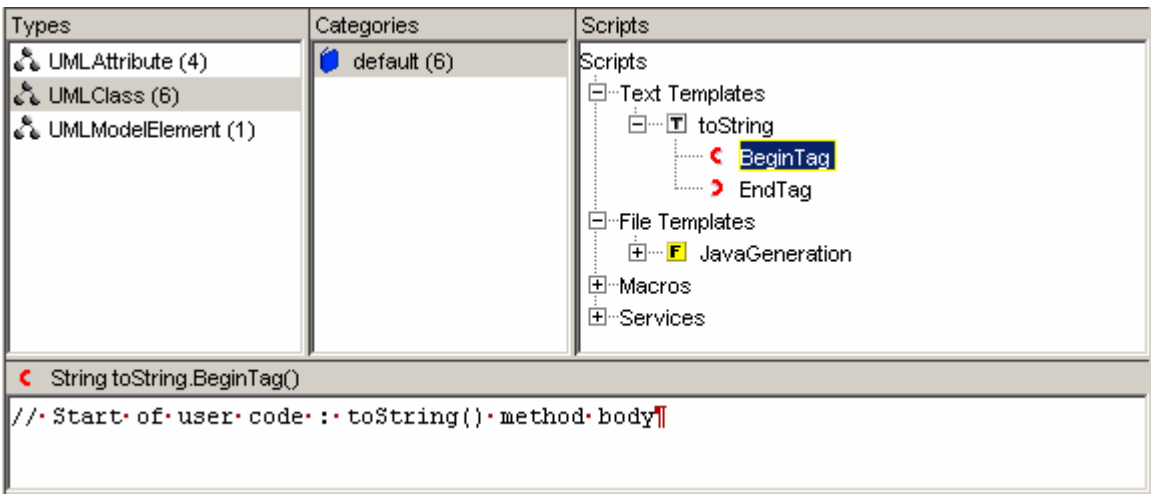


Figure 56 – Unique BeginTag

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Call the BeginTag and EndTag templates in the body of the "toString" template.

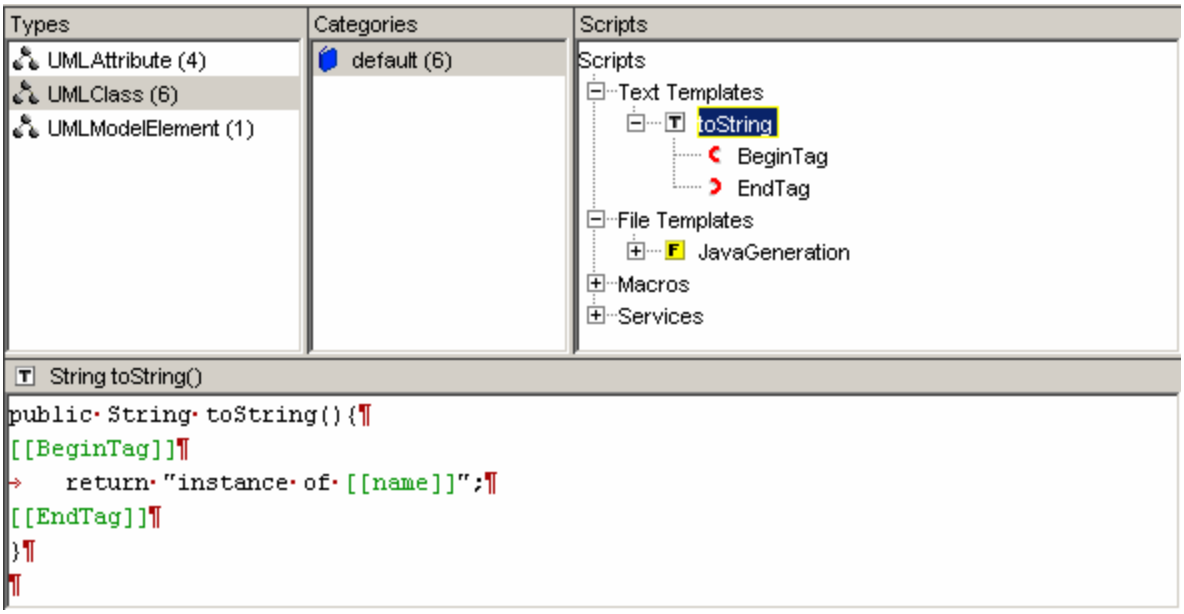


Figure 57 – Modification of the toString template

Go to the **Generation** tab and start the generation, but this time choose to generate into a file.

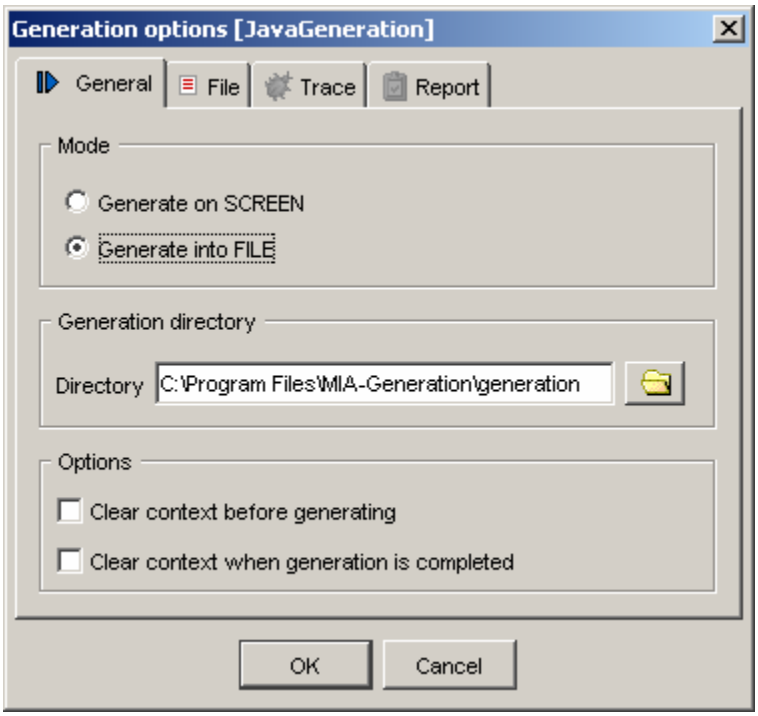


Figure 58 - Generate into file

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The first time that you generate into a file, the file doesn't exist and is created. Therefore there is no possibility for user code, and the default implementation of *toString()* is generated.

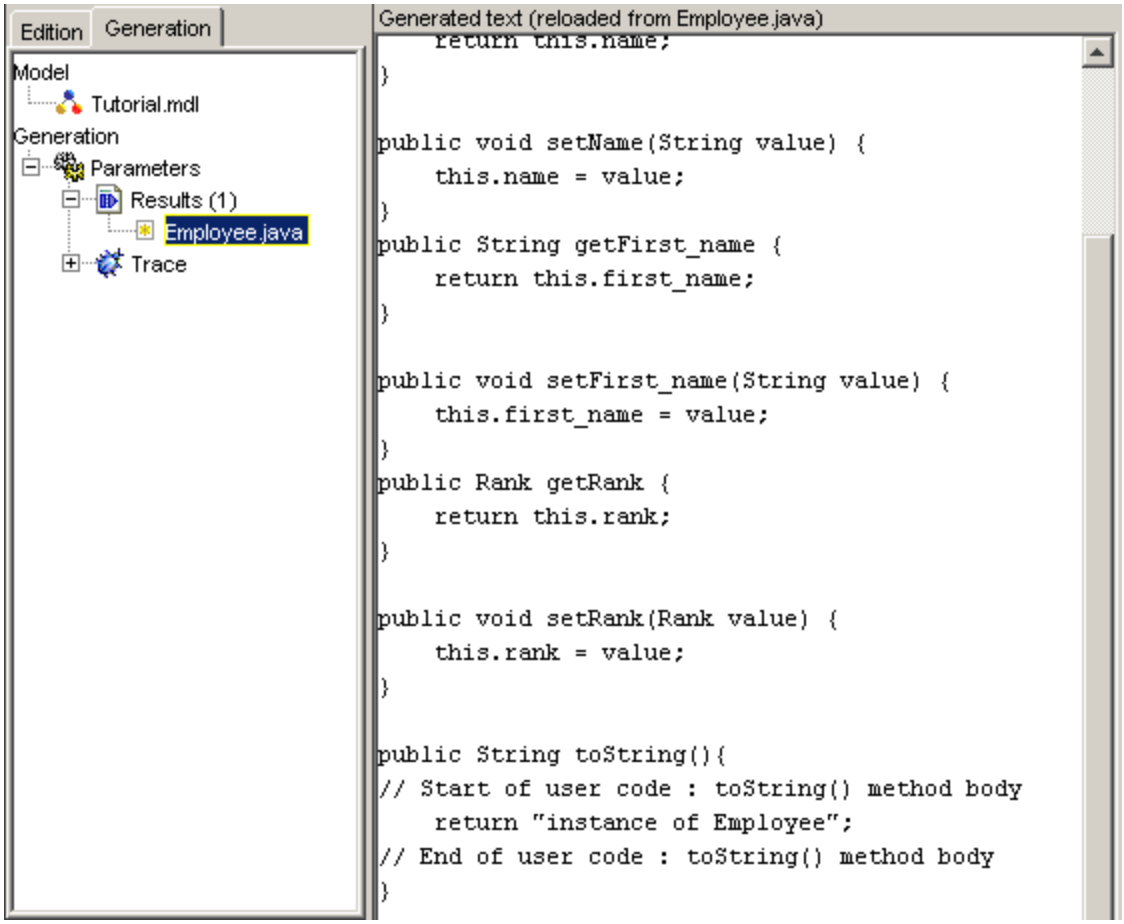


Figure 59 - Generation into a file

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Filename resolution

Once a generation has completed, the generated files are listed in the *'Results'* list. It is then possible to view the contents of each generated file by selecting the file. The contents appear in the textbox to the right of the file list.

If this filename is already absolute, then the filename string is simply returned and used. Otherwise this filename is resolved against the current generation directory.

It is possible to sort results using **View** items from the **Results** context menu. You can change results display with **View > File Name**, **View > Relative path**, **View > Full path** items.

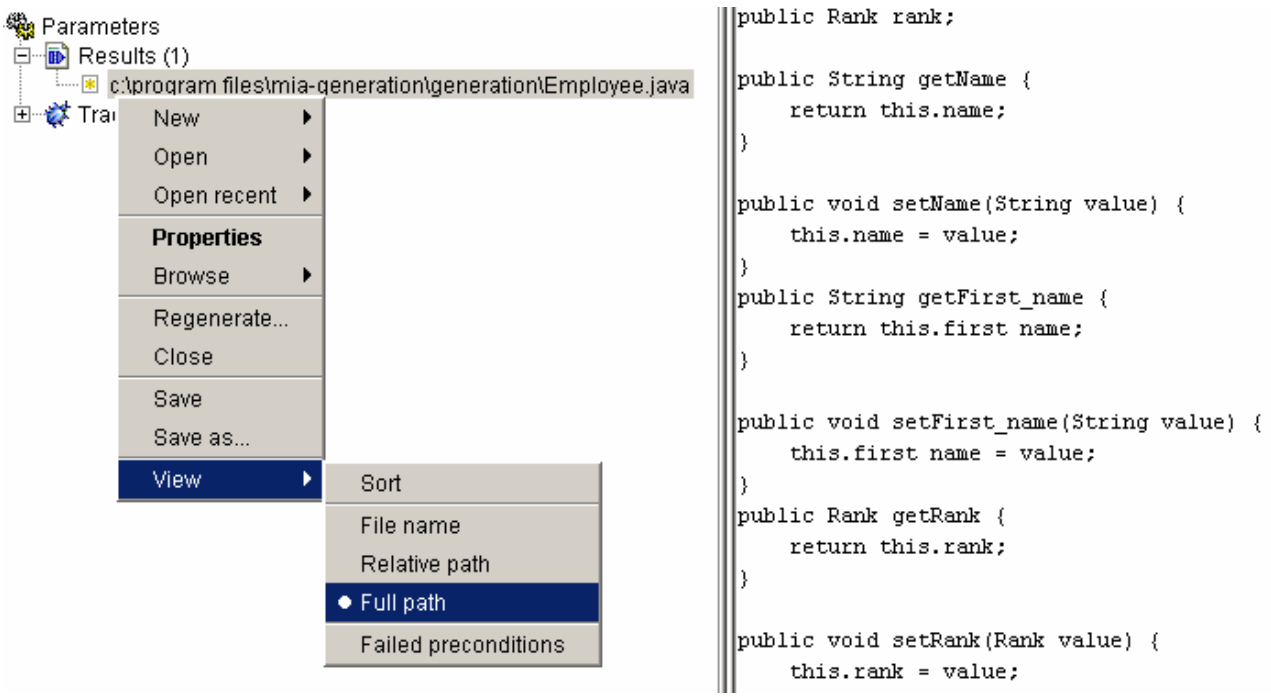


Figure 60 - Sorting generation results

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Edit the generated file with a text editor (e.g. Notepad), and modify the implementation of the method *toString()*, of course without modifying the BeginTag and EndTag.

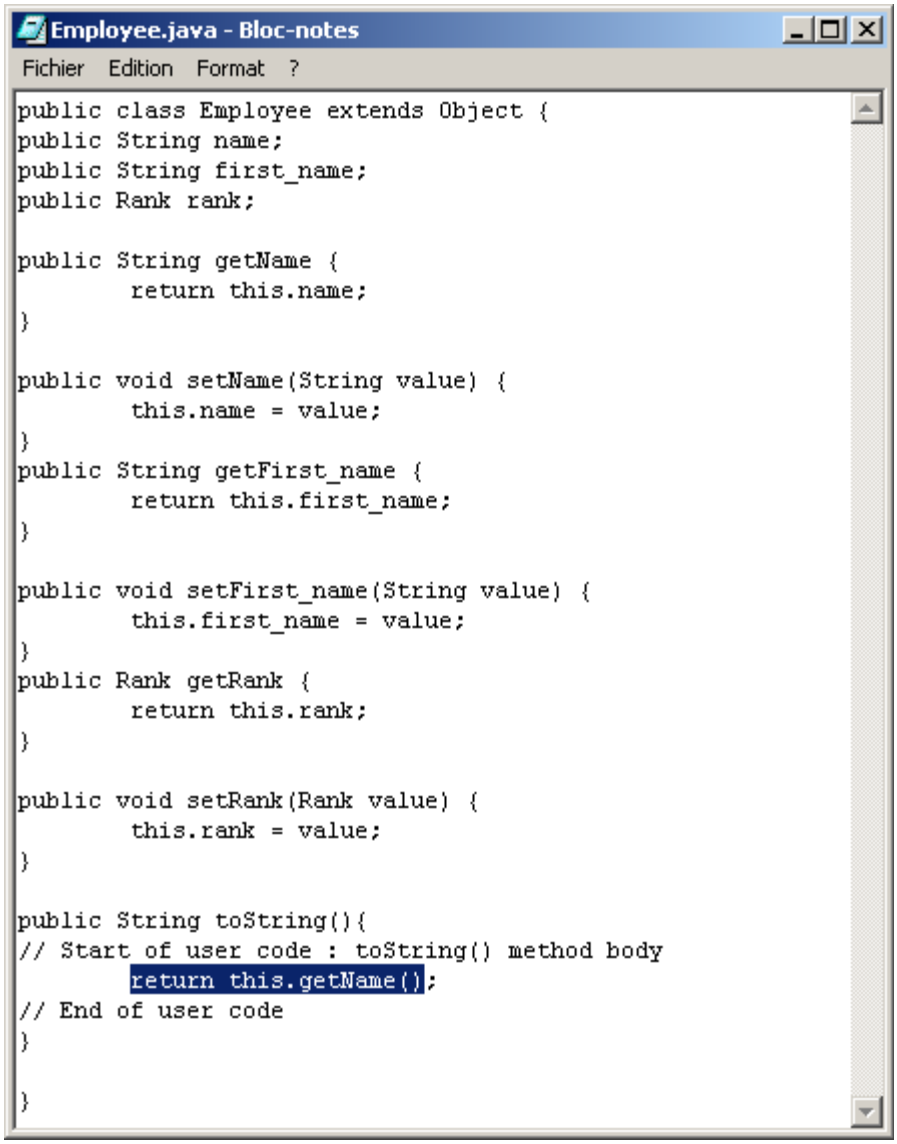


Figure 61 - Manual modification of the generated code

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Regenerate the file using the **Regenerate…** item from the Results context menu.

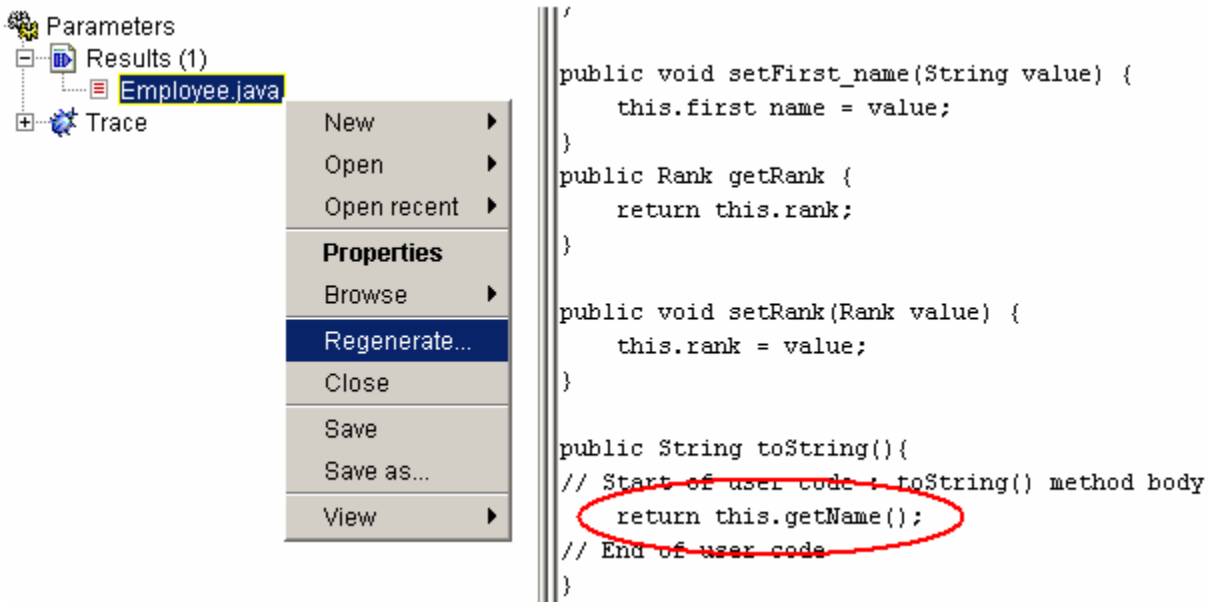


Figure 62 - Generation of a manually modified file

At the time of generation, MIA-Generation detects if the file already exists. It then finds the manually changed portions, and inserts this code during the writing of the newly generated contents.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Generation by Scenario

We are going to make the generation a little more robust.

In the preceding example, each time a model is loaded, we will have to select the classes to generate, and the File Template to evaluate. This means that in order for a user to use the generator, he would have to know which objects to choose in the model, as well as knowledge of the way the generator has been built : he has to know that the generation is done with the “Java Generation” File Template and the objects of type *UMLClass*. We are going to use the notion of a generation **scenario** to make it easier to launch a generation

Creating a scenario

1. Go to the **Edition** panel.
2. Create a public scenario by clicking on the New Scenario button from the toolbar.

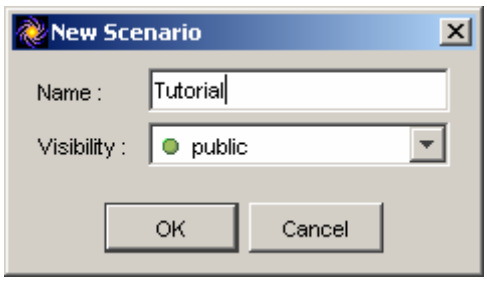


Figure 63 - Creation of a scenario

3. To create a Generation Description, you can click the New Generation Description button from the toolbar.

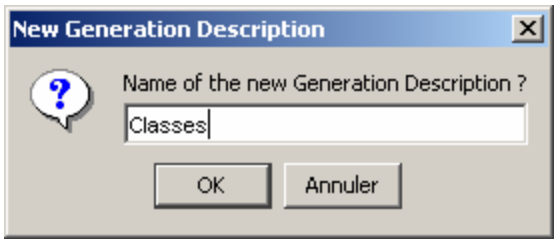


Figure 64 - Creation of a Generation Description

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

A generation description is like a persistant generation unit. Generation description breaks down into two steps : **objects selection** and **file templates choice**. Initially, it is necessary to select the objects to which the generation will apply. Then, the scripts that you want to apply to these objects must be chosen.

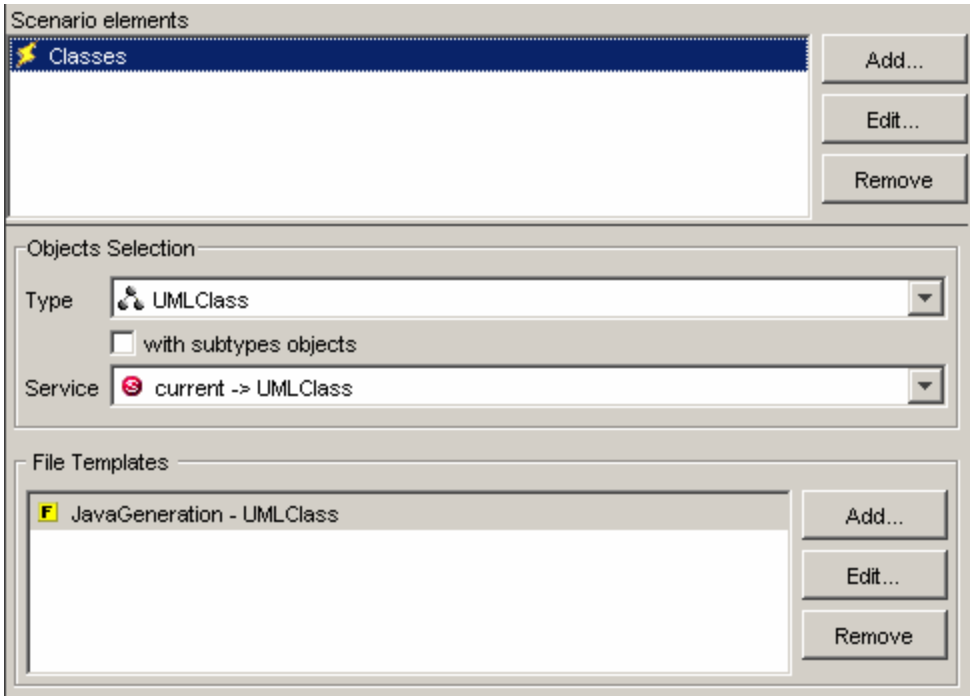


Figure 65 – Generation Description

Objects selection

To select the objects to which the generation will apply, we use a service to filter objects from the model. There is two levels into this objects selection :

1. In the **Type** combo-box, choose the type of first level objects that you wish to select. During generation, all objects of the selected type will be selected. If you also wish to view the objects of the subtypes, select the **with subtypes objects** checkbox.
2. A service must be selected from the ones available for the selected type. Applied to the objects selected with the **Type** combo-box, the service may allow you to derive other objects. By default, the current service (available in the standard version) is selected, which makes it possible to deduce exactly the same objects as those selected with the **Type** combo-box.

This mechanism is very useful when you wish to launch a generation on a precise subset of objects. Rather than selecting them by hand, you can write a service that returns them automatically. In this way, for example, you can easily (and without any risk of omission) launch a generation on all persistent classes of a model, or on all subclasses of a particular class.

Select *UMLClass* type and *current* service to get all classes without filter.

File Templates selection

The selected service is automatically applied to each object of the selected type, and the derived objects are used to evaluate file templates. In the **File Templates** list, there will appear every Files template defined (and loaded) for the **Returned Objects Kind** of the previously selected service.

Select *JavaGeneration* file template and save your scenario using the save button from the toolbar.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

This scenario gives us the possibility to execute the Java generation on all the classes in the loaded model. To start the generation, go to the **Generation** tab and select the **Scenario** panel. Start the generation by clicking on the button "Generate".

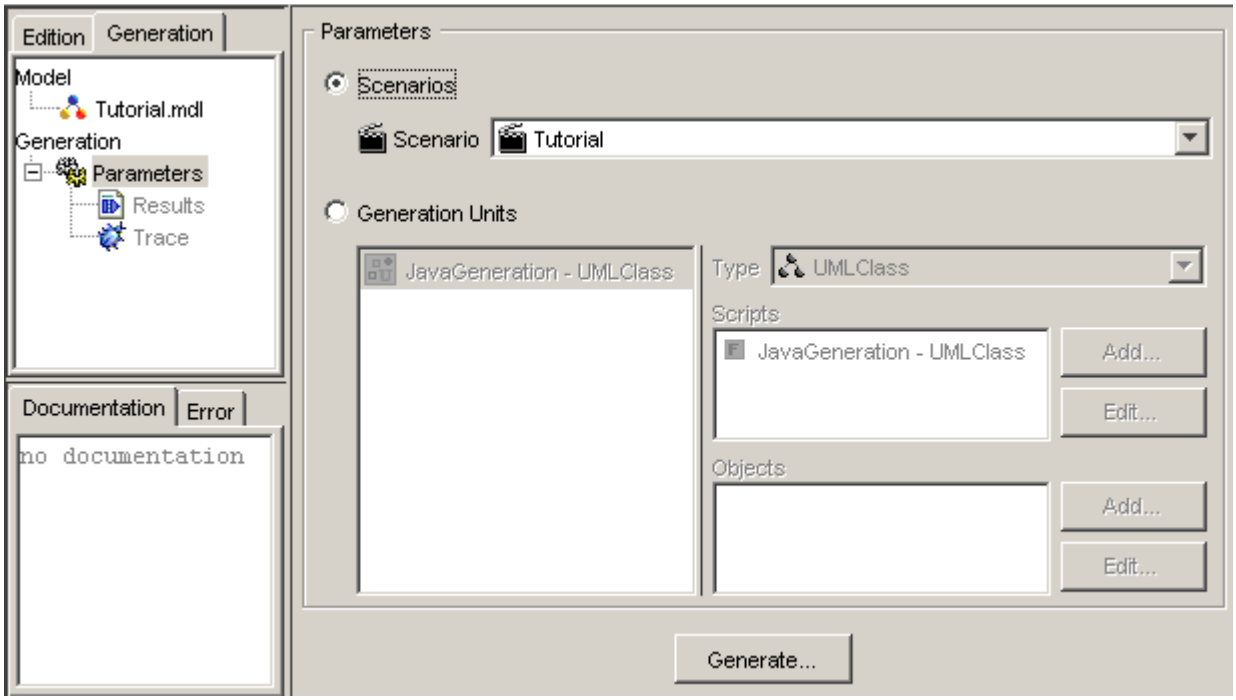


Figure 66 - Generation by scenario

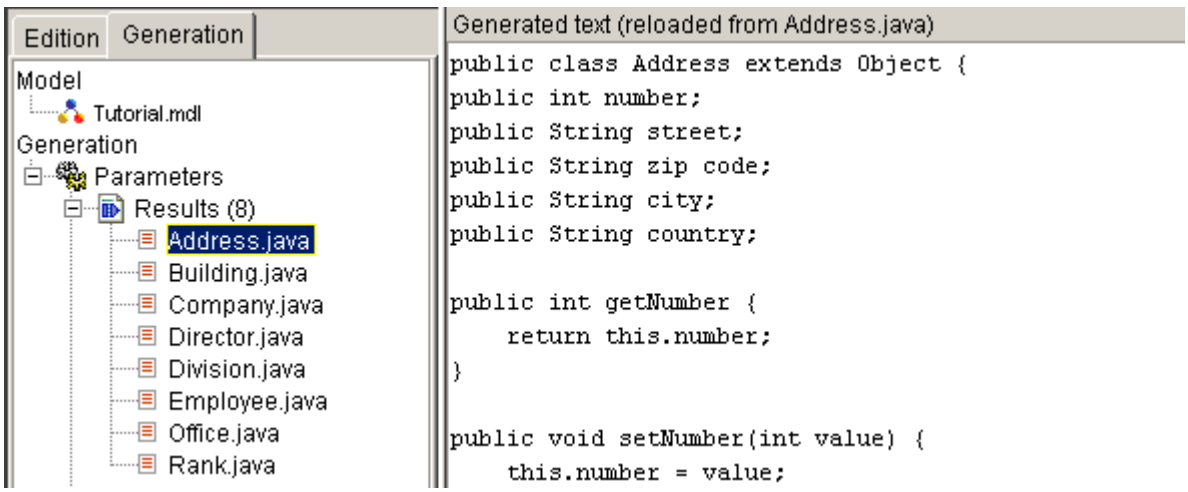


Figure 67 - Results

A scenario can have different units of generation and can have child-scenarios. Therefore we can easily, without knowing how the generator was implemented, generate an entire collection of files. These scenarios will be used in MIA-Generation Developer by endusers.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Using services into scenarios

Services are special macros that enable you to return, not text, but objects from the loaded model. For example, you can write services to recover all abstract classes of a model, or recover all persistent attributes of a class.

With a service you are able to recover objects into scenarios (like you do manually from a generation unit). We are going to create a new service named *allClasses* to get all UMLClass instances.

Getting all classes

Create a new service named *allClasses* on the *UMLPackage type*. Returned objects kind is *UMLClass*.

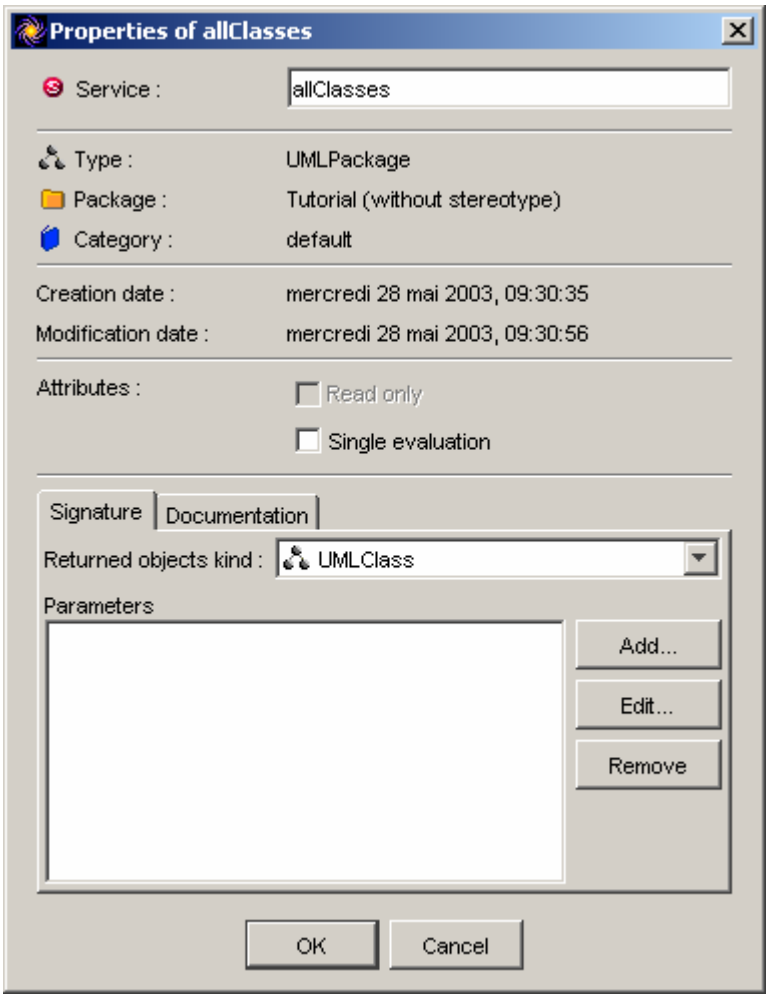


Figure 68 - Creation of a service allClasses

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

This service will use *ownedClassifiers* and *ownedPackage* links defined in UML metamodel for the type *UMLPackage*.

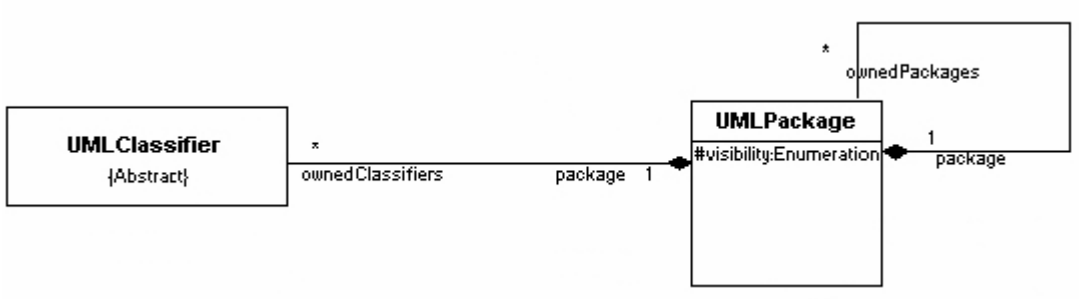


Figure 69 - UMLPackage – UMLClassifier relationships

Note : *UMLClass* is a subtype of *UMLClassifier.*

Modify the text of the *allClasses service :*

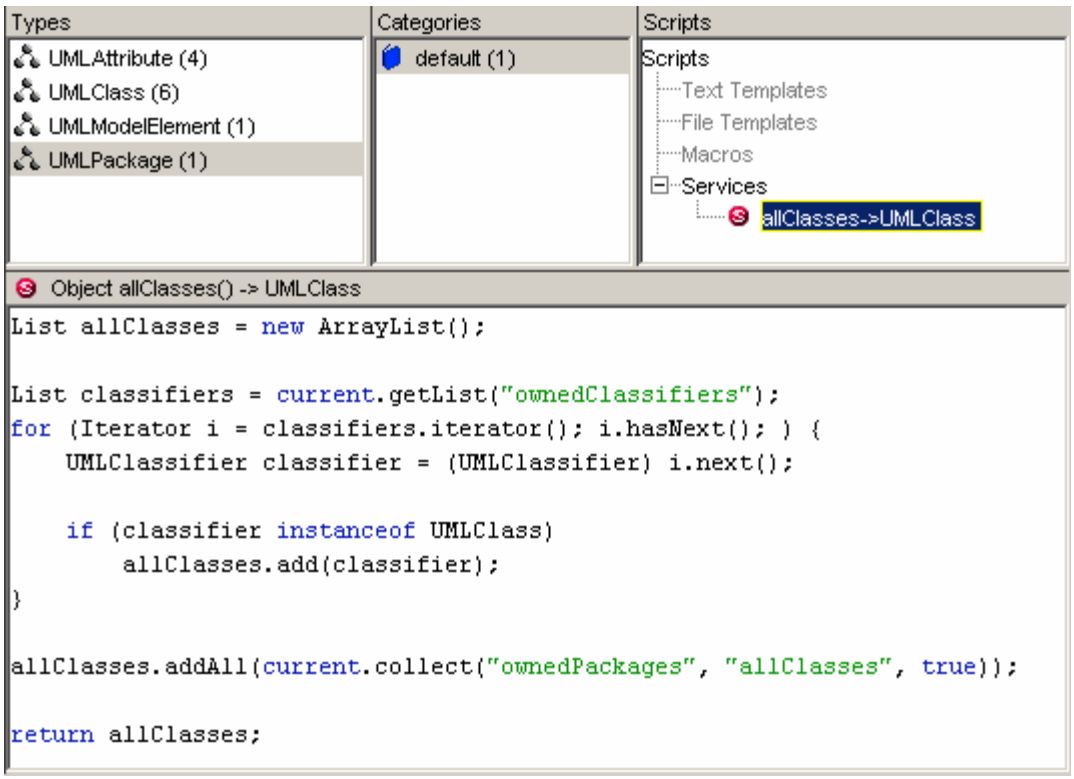


Figure 70 - Service allClasses

With this kind of service, you'll be able to filter model elements. In the *allClasses* service, there is only one filter applied : the test of the type *UMLClass.*

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Calling allClasses service into a scenario

To use *allClasses* service, you must call it from a package. The model is an instance of *UMLModel*, (wich is a subtype of *UMLPackage)* and contains all other packages To explore all packages, you just have to call *allClasses* service on the *UMLModel* instance.

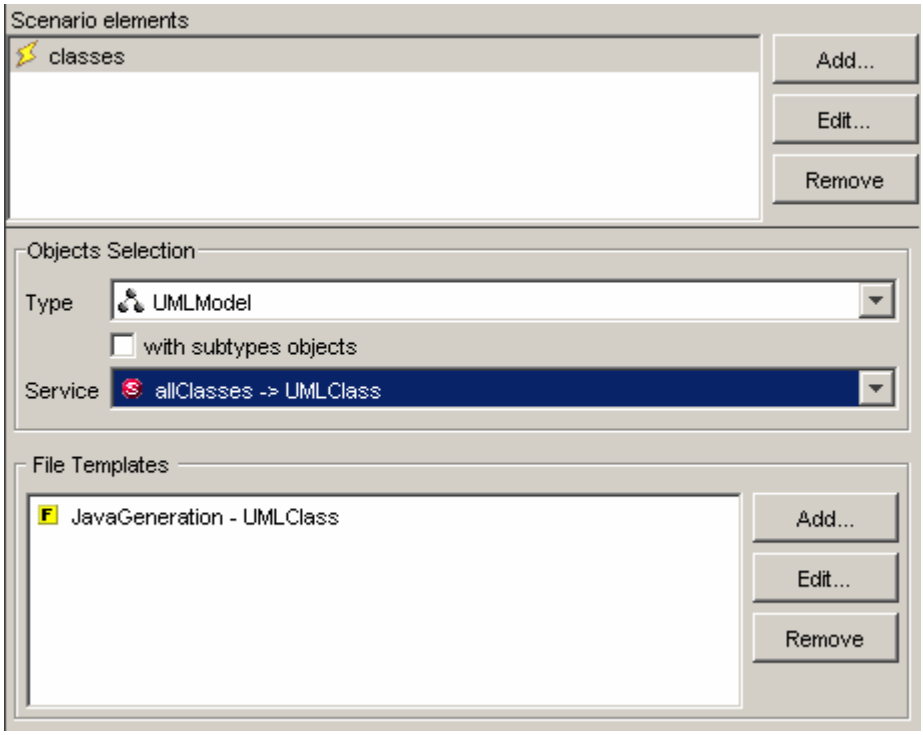


Figure 71 – Tutrial Scenario modified to call allClasses service

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Advanced concepts

Using stereotypes

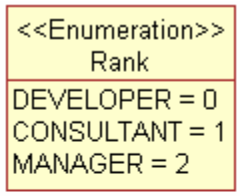
Using stereotypes is a way to customize a model, defining in fact new types. The stereotypes defined on types are available with MIA-Generation because this kind of object is part of the model and different generation choices could be based on the usage of stereotypes.

We are going to use a stereotype <<Enumeration>> in our generator to customize generation of enumerable constants.

Languages like Pascal and C++ let you define a set of enumerable constants like Color=[Red, Green, Blue], which amounts to adding a new data type to the language. These constants can be used for a typesafe index in an array, or used in a loop. Using enumerated constants can make code more readable.

A frequently proposed alternative is the use of Java's "static finals". In Java, such objects are most often known as "enumerated constants."

For example, you might want to define a new data type named Rank with constants DEVELOPER, CONSULTANT and MANAGER as its possible values :




```
public class Rank {
   public static final int DEVELOPER=0;
   public static final int CONSULTANT=1;
   public static final int MANAGER=2;
}
```

Figure 72 - <<Enumeration>> class

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

1. Create a new template named *JavaEnumerationBody*

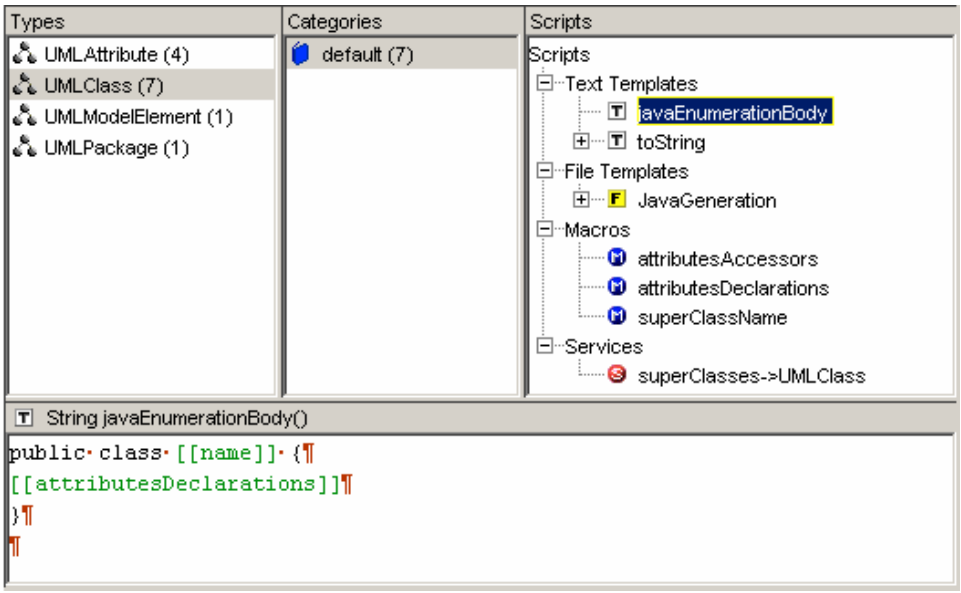


Figure 73 – javaEnumerationBody Template

2. Copy the content of file template *JavaGeneration* to a *JavaClassBody* template on the *UMLClass* type.

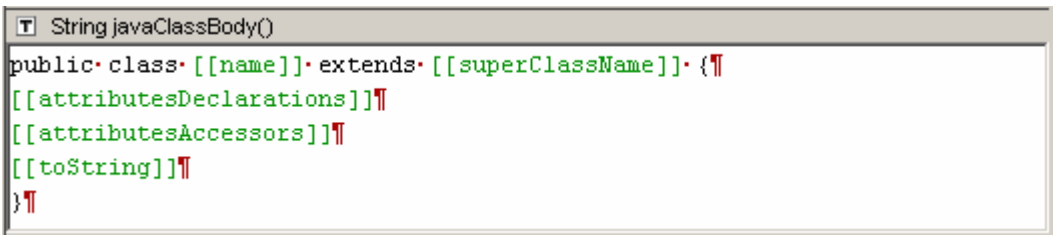


Figure 74 - javaClassBody Template

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

3. We have to test if the current class is an enumeration or not. Create a new macro named *isEnumeration* that returns an *java.lang.Boolean* instance. Select the Boolean type from the **Primitive** combo-box.

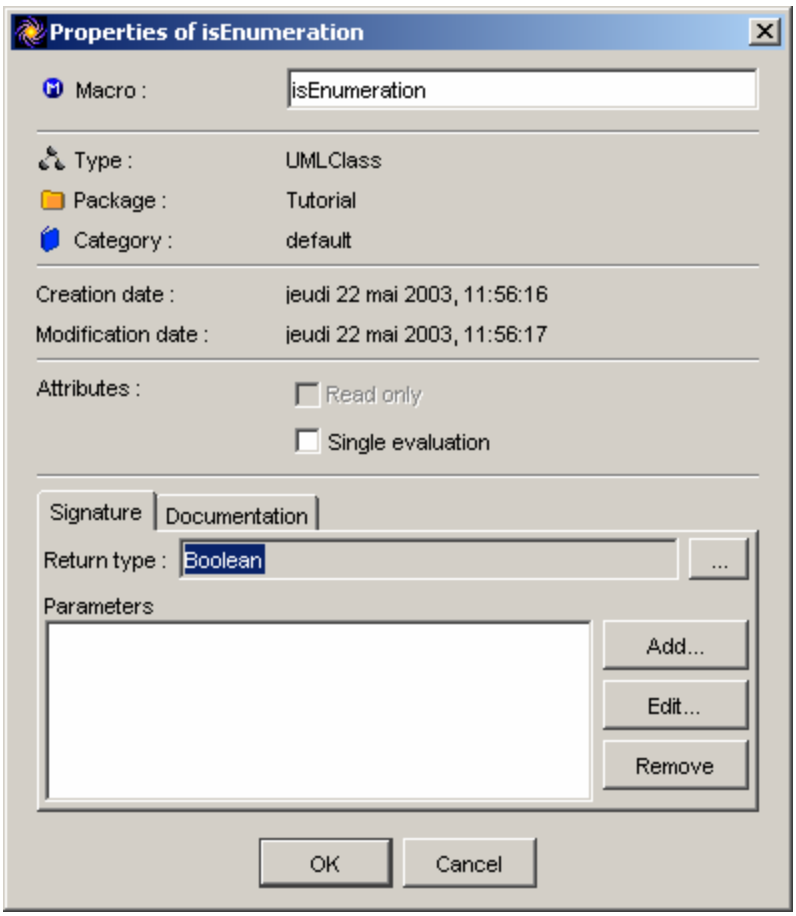


Figure 75 – isEnumeration macro

Modify the text of the *isEnumeration* macro :

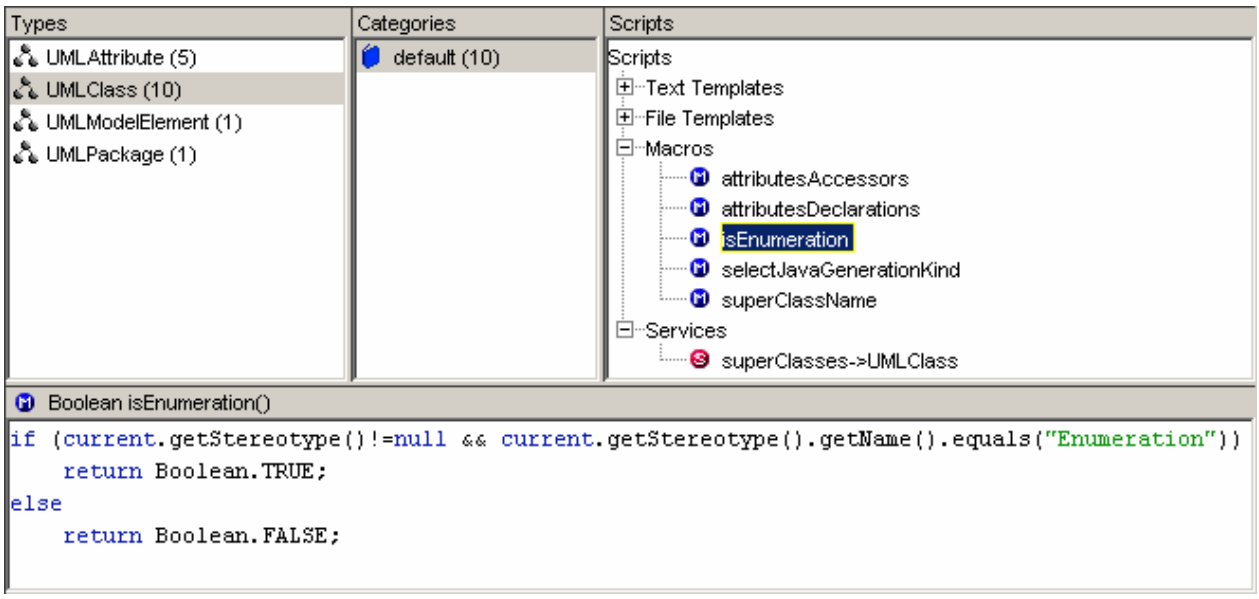


Figure 76 – isEnumeration macro

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

4. Create a new macro named *selectJavaGenerationKind* to make generation choices based on the usage of stereotypes. Modify *JavaGeneration* text to call it.

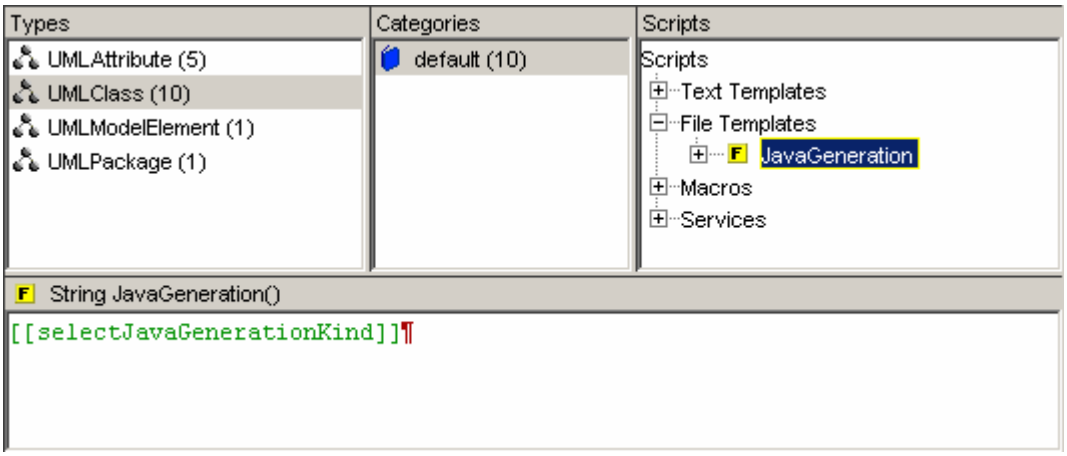


Figure 77 – JavaGeneration file template

The selectJavaGenerationKind is the following :

Figure 78 – selectJavaGenerationKind Macro

Note :

In MIA-Generation, scripts return Object. MIAObject provides several useful methods for converting objects to values of the primitive type.

getBoolean() invokes a script matching the specified call, a script whose evaluation result represents a boolean value. It returns `true` if the evaluation result is `Boolean.TRUE` or it is equal, ignoring case, to the String "true" (using `Boolean.valueOf(String)`).

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Attributes of an <<Enumeration>> class must be not generated like those of standard Java™ class. Modify the *attributesDeclarations* macro on *UMLClass* to branch attributes generation.

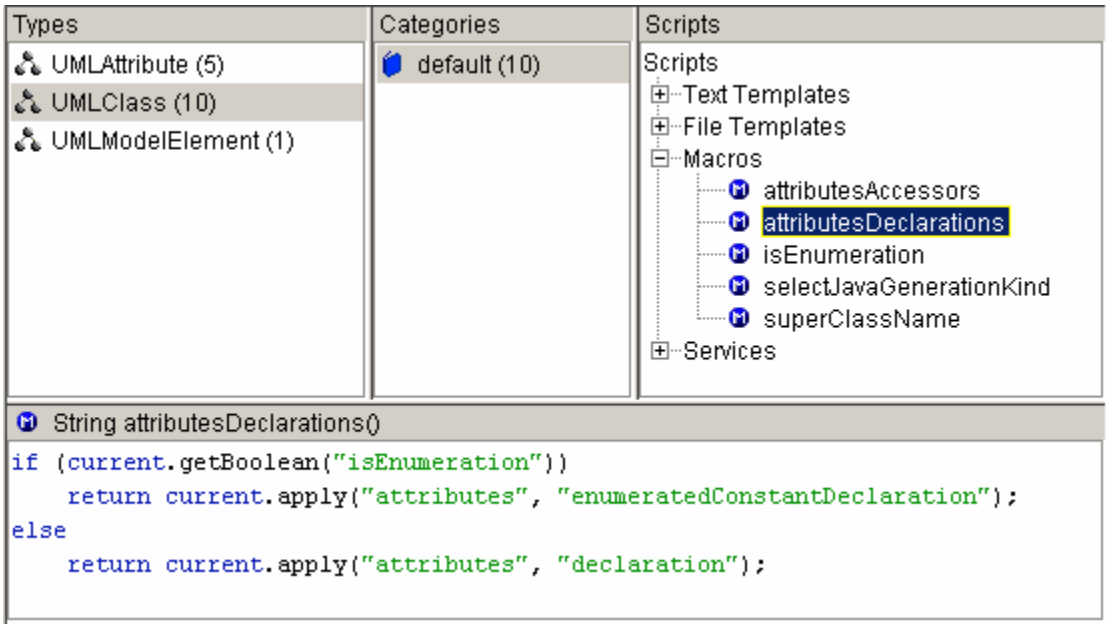


Figure 79 - attributesDeclaration Macro

5. Create a new template named *enumeratedConstantDeclaration* to declare Java's "static finals". You can press **CTRL+C** and **CTRL+V** to get a copy named *declaration1* from the template named *declaration*. Rename it using **Rename** or **Properties** context menus from the Scripts list.

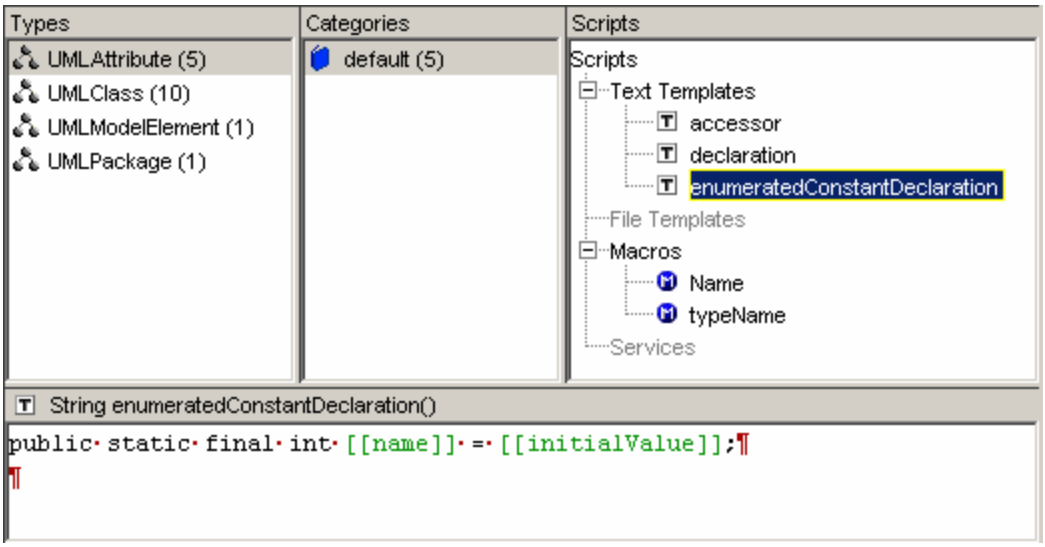


Figure 80 – Enumerated Constant Declaration

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

6. Test your modification on a new Generation Unit *JavaGeneration – UMLClass.* Select *Rank* and *Adress* objects using the service *allClasses* on *UMLModel*.

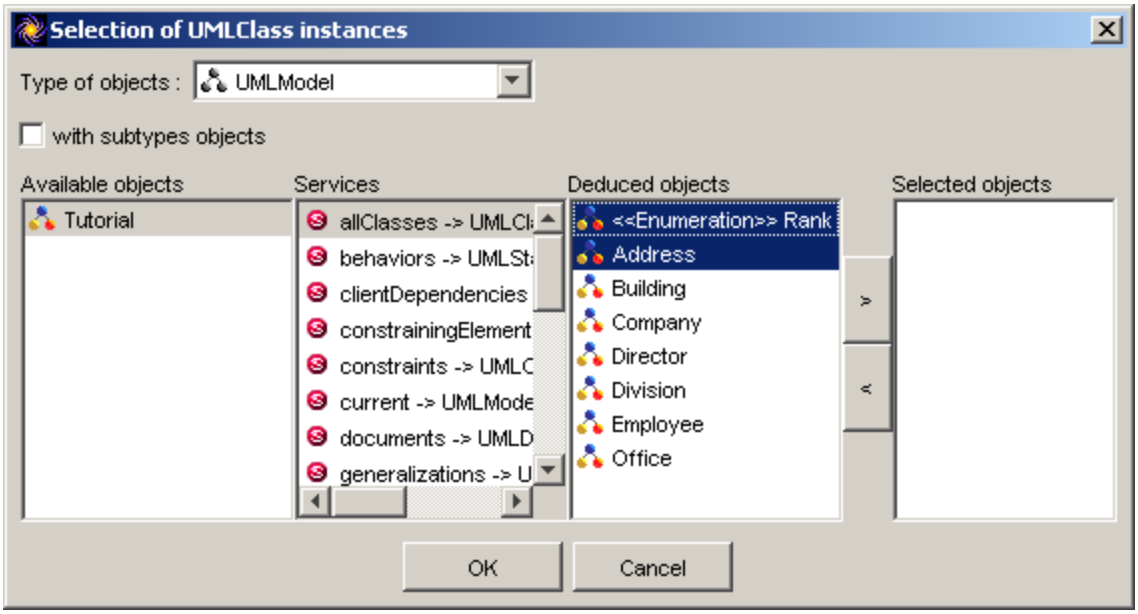


Figure 81 – Rank and Address selection

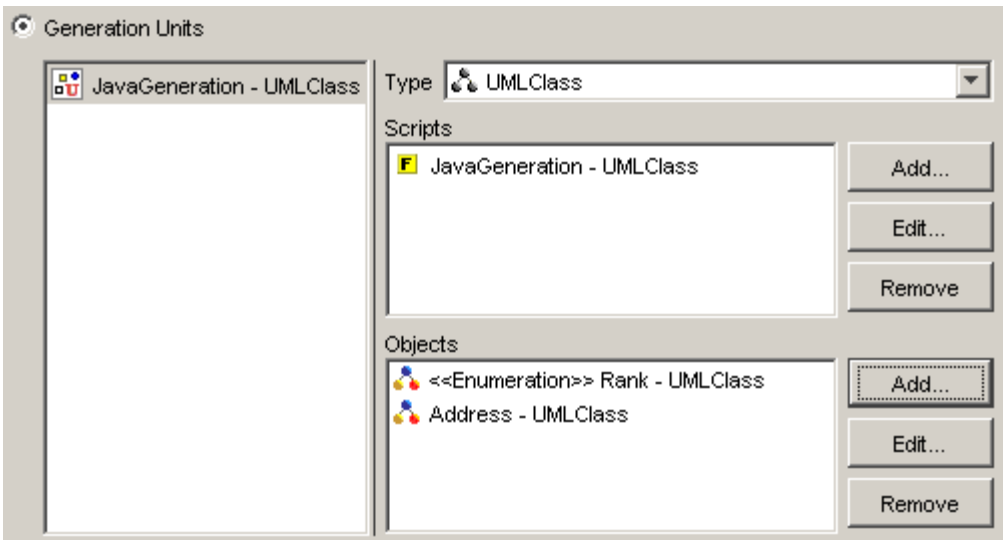


Figure 82 – "JavaGeneration – UMLClass" Generation Unit

Use the **Trace** mode to see how stereotypes are used to make generation choices in the *selectJavaGenerationKind* and *attributesDeclaration* macros.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

For the class *Rank* :

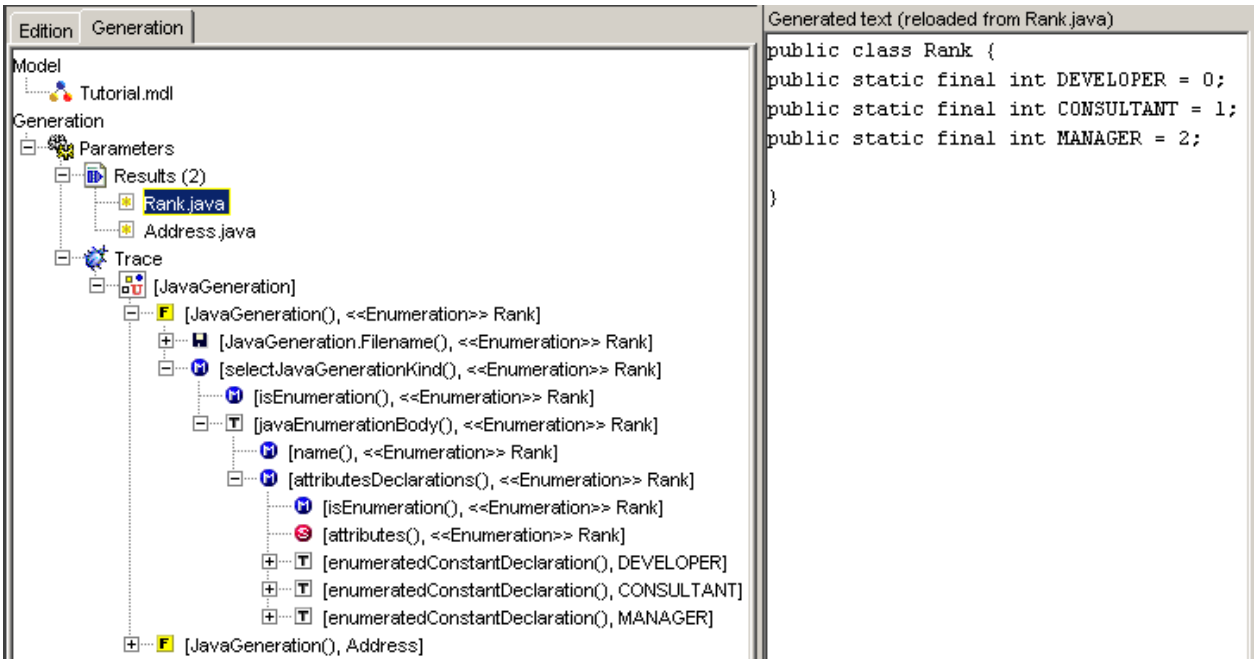


Figure 83 - <<Enumeration>> Rank

For the class *Address* :

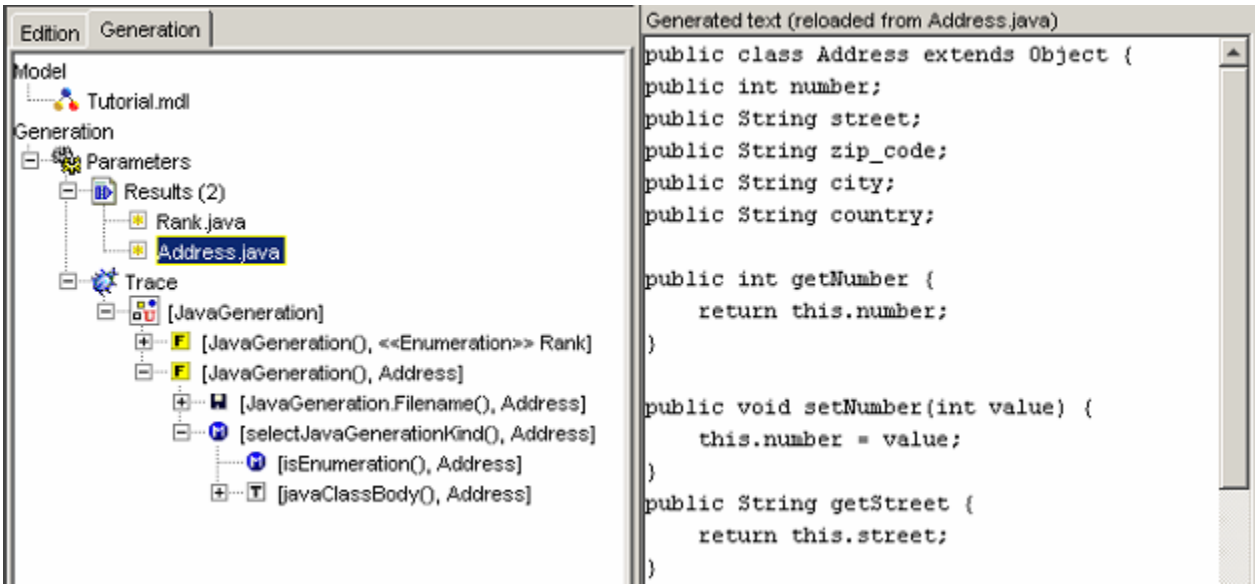


Figure 84 - Address

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Using profile

With stereotypes, many scripts will have to check the stereotype of the model elements, to foward the generation using another script, more appropriate to this stereotype. For example, a class could be stereotyped "Table", "Enumeration" or "Business" to denote a mapping to database concepts, an enumeration definition or a business entity and scripts would look like this :

```
if (stereotypeName.equals("Table"))
       ... // Tables
else if (stereotypeName.equals("Enumeration"))
       ... // Enumeration
else if (stereotypeName.equals("Business"))
       ... // Business
else
       ... // Other classes
```

In that case, it would be more interesting to use a **profile**. A profile enables to customize a metamodel, by adding new types. It is a set of stereotypes, each stereotype defining a new type.

MIA-Generation supports the concept of profile, and extends its use to the meta-model : the concept of Profile can be seen like a means of extending a meta-model easily. The concept of profile, introduced with UML 1.4, is a means to extend UML. A profile is specified by a list of UML types to use, a list of stereotypes, a list of tagged-values, and a list of constraints.

Each user can define his own list of new classes. He will then be able to define scripts directly on these classes, and the imported models will be able to take account of these classes at the time of instantiation of the objects.

Implicitly, a stereotype is a way to define a new type of object. Thus, MIA-Generation provides the possibility to manipulate the stereotyped objects like objects of a specific type.

For instance, with all of the classes that have a stereotype <<*Table*>>, MIA-Generation can instantiate a instance of a type *Table, a* subclass of *UMLClass*, rather than just an instance of *UMLClass*.

This enables to use polymorphism rather than "if" statements.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creation of a profile

We are going to create a profile to manage our stereotype *<<Enumeration>>*.

1. Create a new profile using the New Profile button from the toolbar. Specify the name of our profile : "Tutorial"

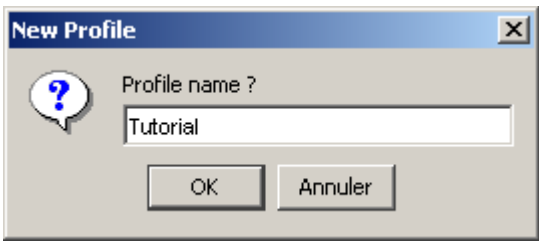


Figure 85 – Creation of a profile

2. The function **Add…** from Profile Editor context menu adds a subtype to the selected type (either a base type of the meta-model or a type of the profile). Add a type named "Enumeration" to *UMLClass.*

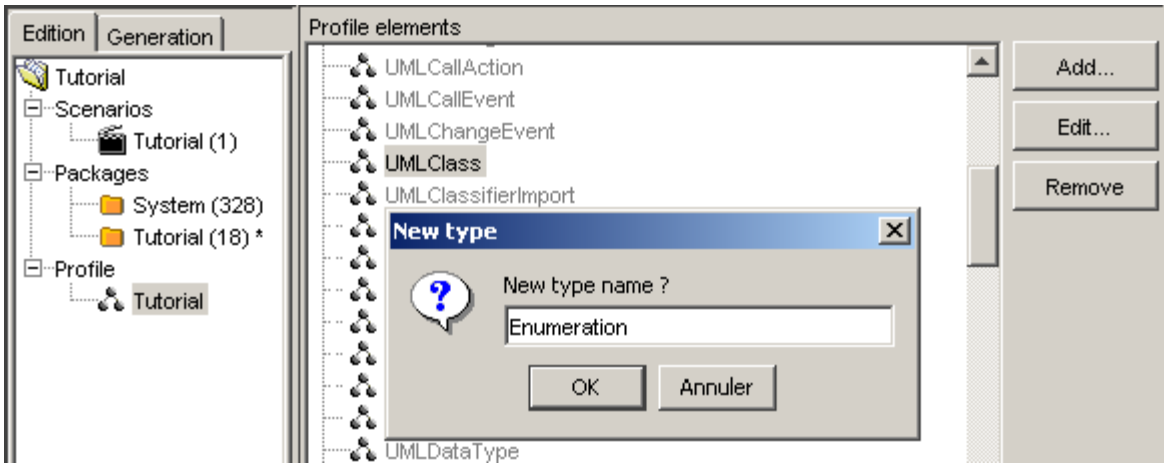


Figure 86 – Creation of a type

3. It is possible to only show the types of the meta-model that are extended by the profile (context menu **View >Mask types without profile**).

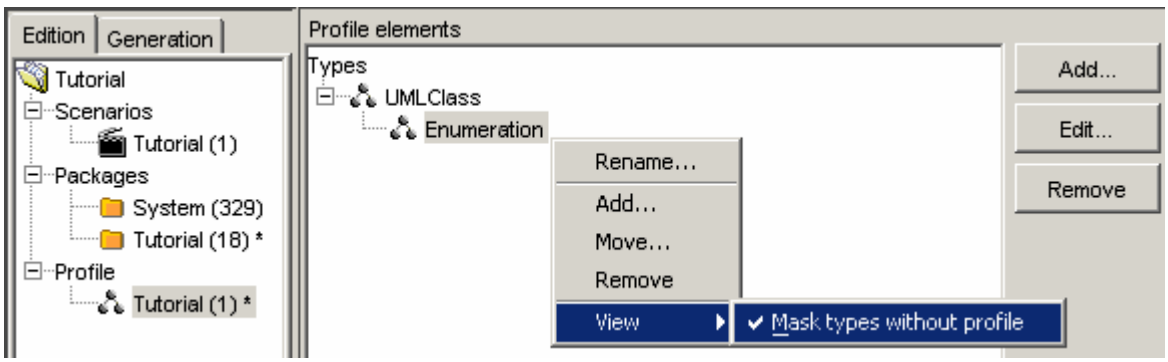


Figure 87 - Enumeration type

4. Save the profile and the project using the Save button from the toolbar. Project will reference your profile and it can be restored at a later time.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Creating scripts on the profile types

The types defined in the profile appear in the Types tree. You can then define scripts on these types exactly the same way as those for the base types of the meta-model, and benefit from the same functionality (inheritance, polymorphism, etc…).

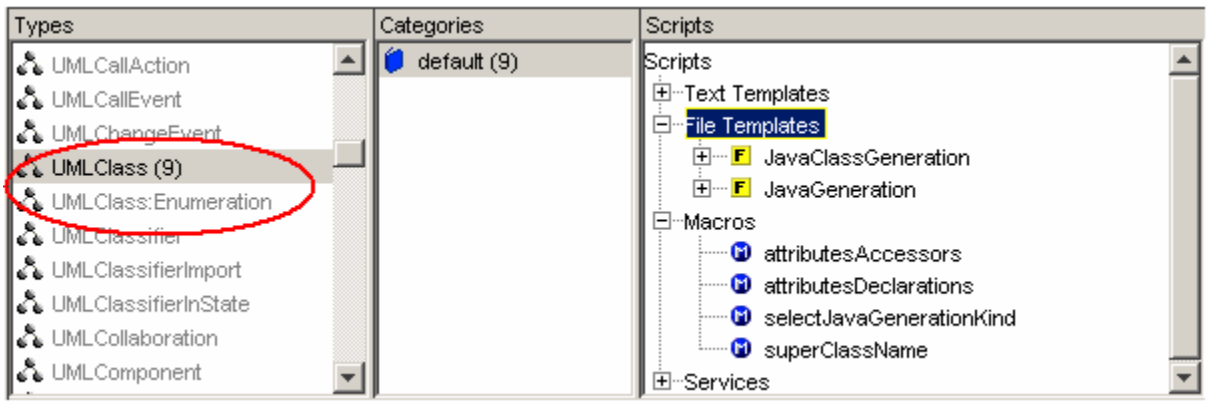


Figure 88 - Profile type in the Types tree

Reorganizing scripts

This tutorial provides a step by step walkthrough of the scripts reorganizing to manage profile.

1. Delete the macro *selectJavaGenerationKind*. You can use DEL key or **Remove** context menu from the Scripts list.
2. Copy the content of template *javaGenerationBody* into the file template *JavaGeneration* on *UMLClass.* Then, delete the template *javaGenerationBody*.
3. Move the template *javaEnumerationBody* on the type *UMLClass:Enumeration* and rename it into *JavaGeneration* (**Refactor > Move …** from the Scripts list).

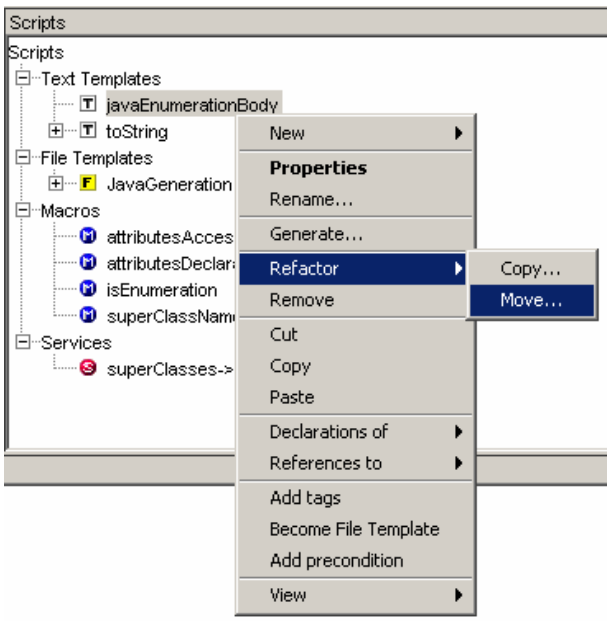


Figure 89 - Refactor menu

Check the **rename** option and rename it into "JavaGeneration".

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

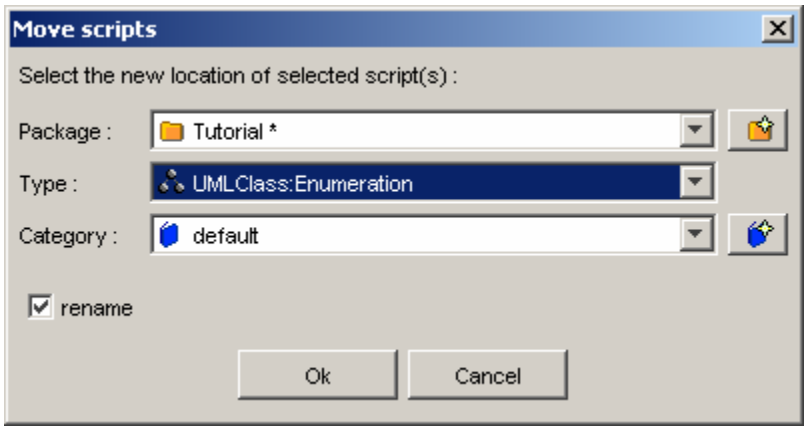


Figure 90 - Move scripts Dialog Box

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

4. Use **Become File Template** context menu (from the Scripts list) to change template *JavaGeneration* into file template :

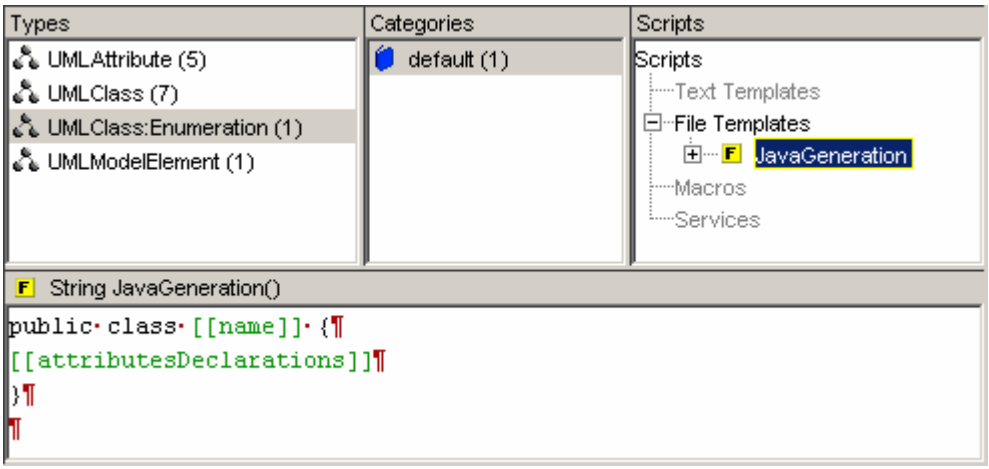


Figure 91 - File Template JavaGeneration on UMLClass:Enumeration

5. Delete the stereotype test into the macro *attributesDeclaration* :

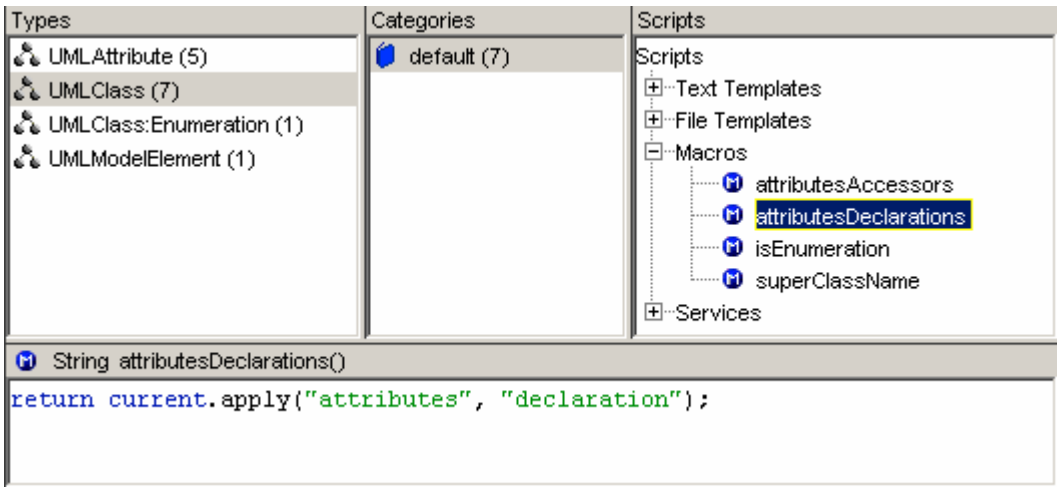


Figure 92 - Macro attributesDeclaration on UMLClass

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

6. Copy this macro *attributesDeclarations* on the *UMLClass:Enumeration* using **Refactor > Copy …** from the Scripts list context menu or **Copy**-**Paste** functionnality. Modify the content of the copied script to call the template *enumeratedConstantDeclaration* on each attribute of the enumeration.

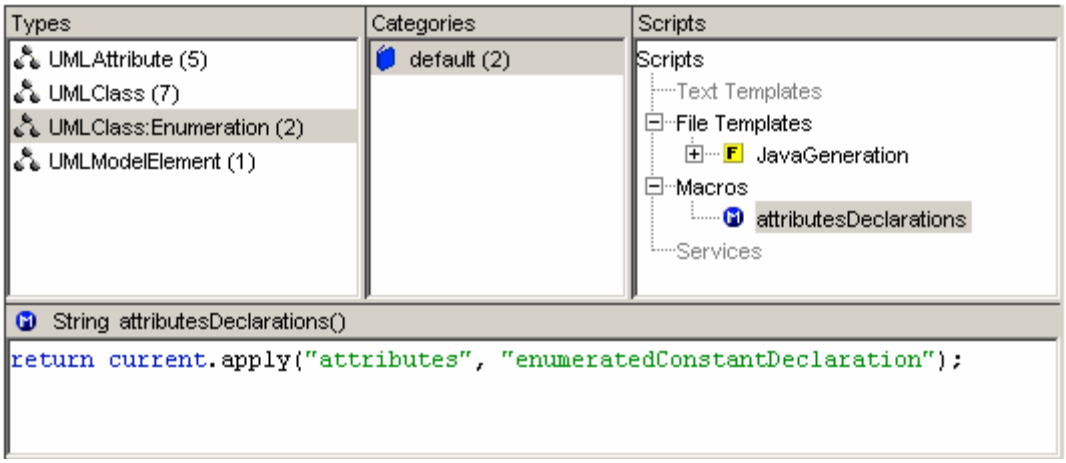


Figure 93 - Macro attributesDeclaration on UMLClass:Enumeration

7. Update *Filename* properties for the two file tempates *JavaGeneration*.

Figure 94 - Filename of file templates JavaGeneration

We have finished scripts modification. We are now going to test it on Rank and Address objects.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

You can see that new scripts defined on *UMLClass:Enumeration* are correctly used to generated enumerated constants declaration.

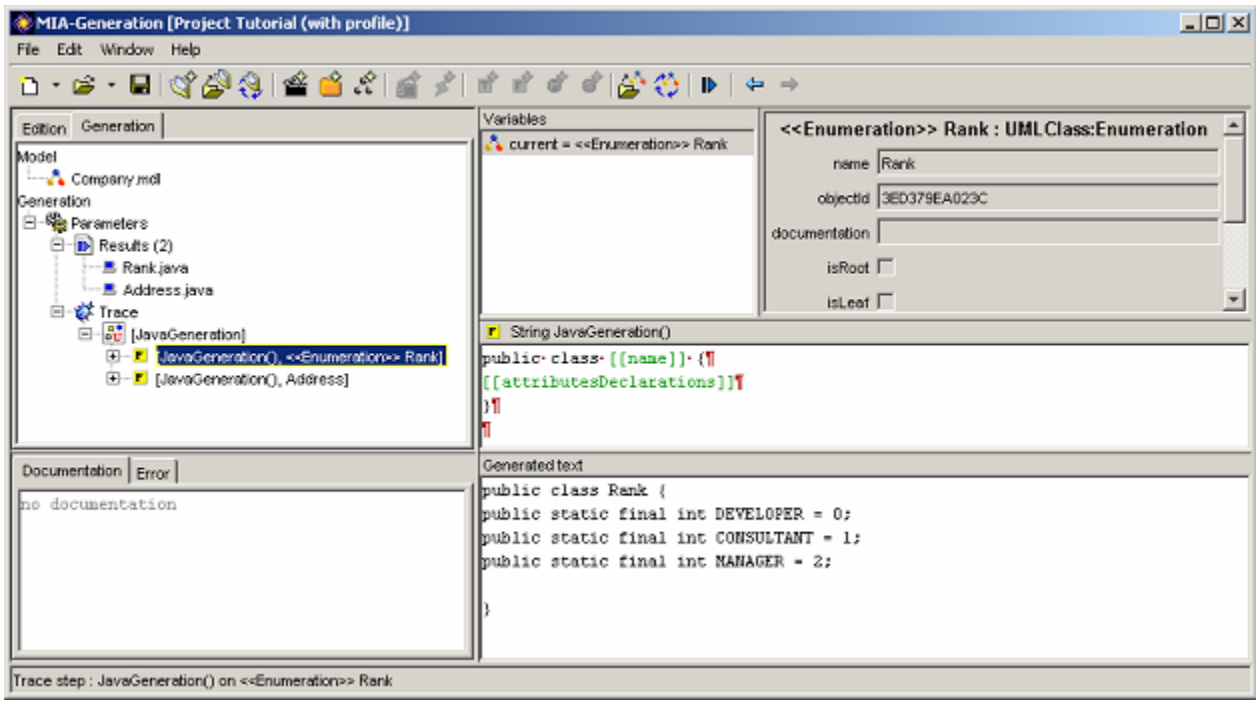


Figure 95 – JavaGeneration Trace

Profile enables to use polymorphism rather than "if" statements. Polymorphism is used to call the file template *JavaGeneration* defined for the real type of the evaluated object : *UMLClass* or *UMLClass:Enumeration*.

If you want to generate *Enumeration* and *UMLClass* in the same scenario without using the service *allClasses*, you will need to check **with subtypes objects** because Enumeration is a subtype of *UMLClass*.

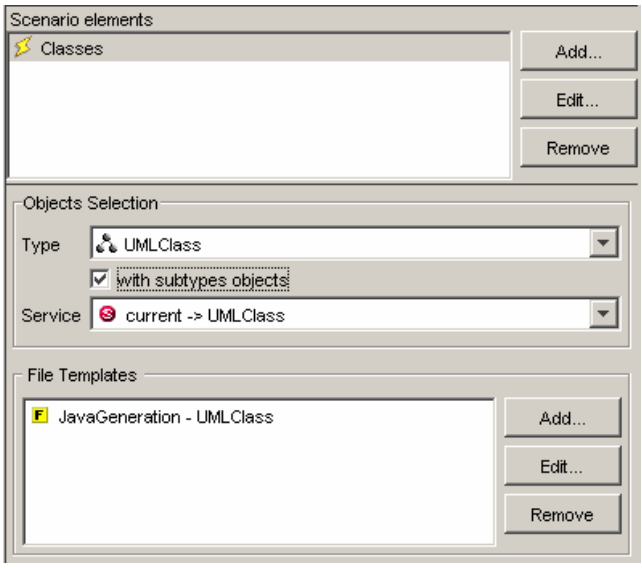


Figure 96 – Modified Scenario

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Coding conventions, patterns and frameworks

The design choices can be translated into coding conventions, patterns and frameworks. The architecture of MIA-Generation is completely open. It is entirely centered on one need : to connect the model resulting from the analysis phase to the generation scripts that reflect the design choices. By means of this connection, MIA-Generation generates the files.

Coding conventions and patterns

Coding conventions can be translated into generation scripts. If the code is generated, it is possible to ensure a single presentation format, which allows the code to be more readable, and thus easier to maintain. For example, getters and setters are coding conventions you can used to write your generator.

In the previous examples, you have used a simple pattern : the use of Java's "static finals", known as "enumerated constants." .

Still, there are a couple of drawbacks to using static final integers. The major drawback is the lack of type safety. Any integer that is calculated or read in can be used as a "Rank" regardless of whether it makes sense to do so.

We are going to see step by step how to build a generator from a prototype implementing a pattern "Safety Enumeration" for Java's enumerable constants. The process used could be applied for any other kind of files you need to generate.

The goal is to define a variable of type Rank so that you get a compilation error rather than a runtime error whenever an invalid value is assigned to that variable. An elegant solution is to use a static final Class with private constructor. This implementation is a variation of the Singleton pattern that limits the class to a predefined number of instances.

Load the project saved into the file "`Tutorial_Pattern_Start.prj`". This project contains an initial generator we are going to upgrade.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

First of all, you need to find the variable part of the generated script :

```
import java.util.*;

public final class Color {

  public static final Color RED = new Color("RED");
  public static final Color GREEN = new Color("GREEN");
  public static final Color BLUE = new Color("BLUE");

  private String id;
  public final int ord;
  private Color prev;
  private Color next;
  private static int upperBound = 0;
  private static Color first = null;
  private static Color last = null;

  private Color(String anID) {
      this.id = anID;
      this.ord = upperBound++;
      if (first == null)
          first = this;
      if (last != null) {
          this.prev = last;
          last.next = this;
      }
      last = this;
  }
  public static Iterator elements() {
      return new Iterator() {
          private Color curr = first;
          public boolean hasNext() {
              return curr != null;
          }
          public Object next() {
              Color c = curr;
              curr = curr.next();
              return c;
          }
          public void remove() {
              throw new UnsupportedOperationException();
          }
      };
  }
  public String toString() {
      return this.id;
  }
  public static int size() {
      return upperBound;
  }
  public static Color first() {
      return first;
  }
  public static Color last() {
      return last;
  }
  public Color prev() {
      return this.prev;
  }
  public Color next() {
      return this.next;
  }
}
```


This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

The script editor is WYSIWYG (What You See Is What You Get). Therefore you only need to **type** the code to be generated (which is littered with macro names) instead of **programming** it. The creation of scripts is thus largely facilitated, since it is possible to conceive them by copying and pasting from existing code for a class.

1. Copy the content of `Color.java` into the template *JavaGeneration* on *UMLClass:Enumeration.*

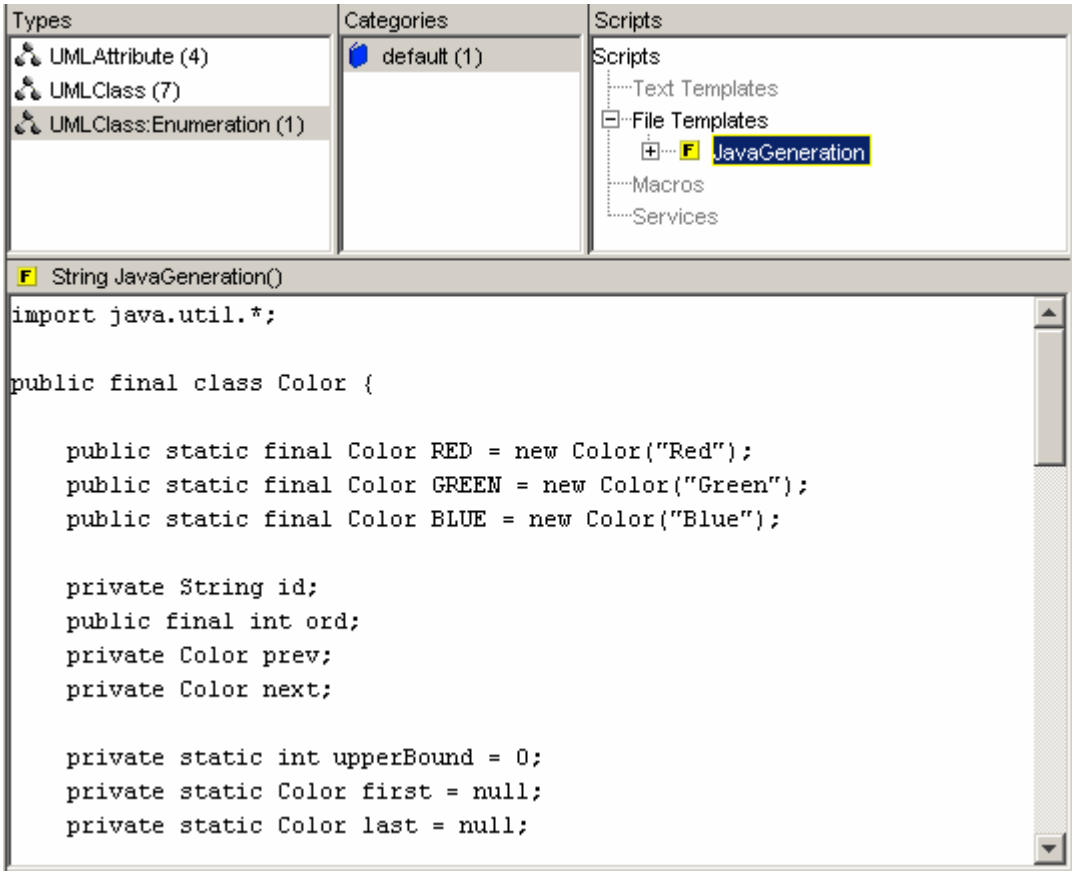


Figure 97 - File Template JavaGeneration

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

2. There is a declaration line for each color "RED", "GREEN", "BLUE". The relation "*One declaration for each attribute of the enumeration class*" means you must create a script on *UMLAttribute* that will be call from the *UMLClass* with an apply(). Cut attributes declaration lines and copy them into a template named "enumeratedConstantsDeclaration" on UMLAttribute.

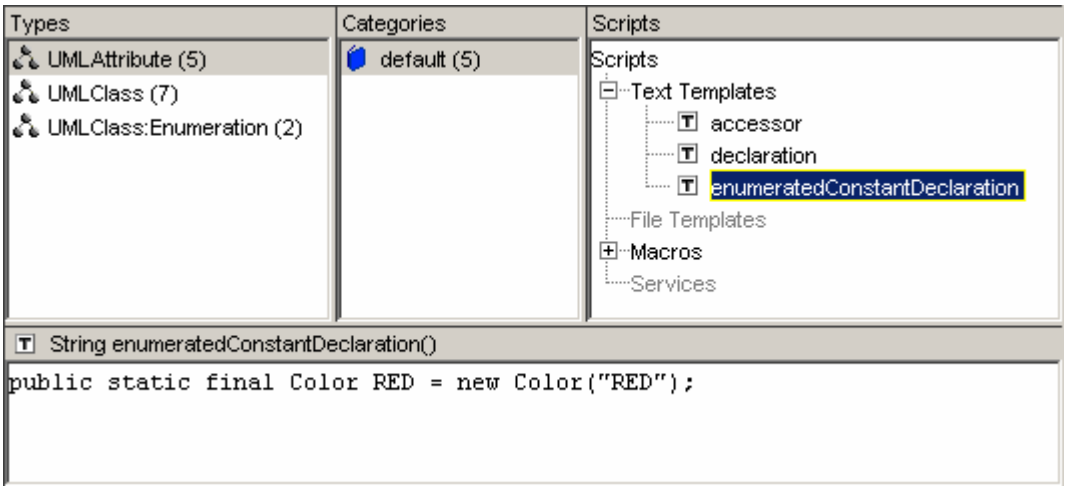


Figure 98 - Template enumeratedConstantDeclaration on UMLAttribute

Save the text after removing "BLUE" and "GREEN" lines. You can see there are two informations to get from model at runtime : the enumeration's name ("Color") and the attribute's name ("RED").

```
public static final Color RED = new Color("RED");
```

Modify the text to call a new macro *ownerName* which get the owner's name and the macro *name* :

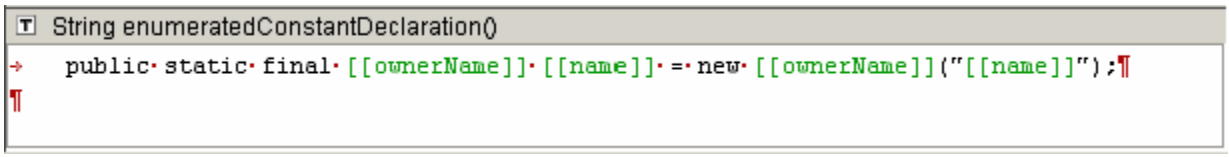


Figure 99 - Template enumeratedConstantDeclaration on UMLAttribute

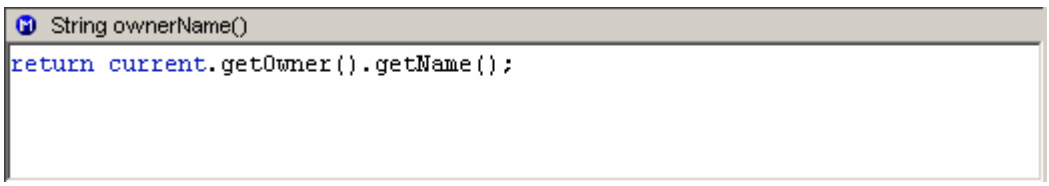


Figure 100 - Macro ownerName on UMLAttribute

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Replace the text "Color" with "[[name]]" in the file template *JavaGeneration* **Find/Replace…** context menu from the text area. Call the macro *attributesDeclaration* to generate attributes declaration.

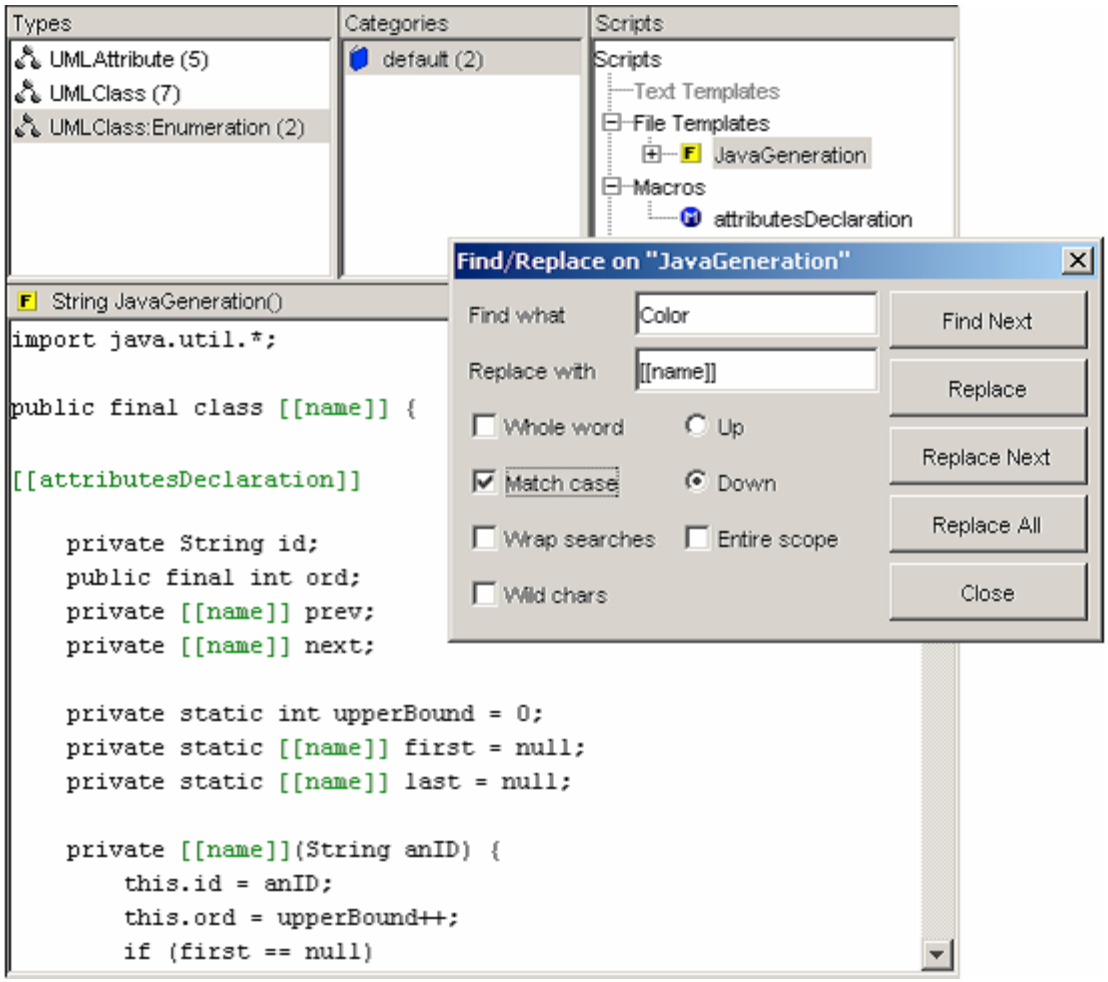


Figure 101 - File Template JavaGeneration

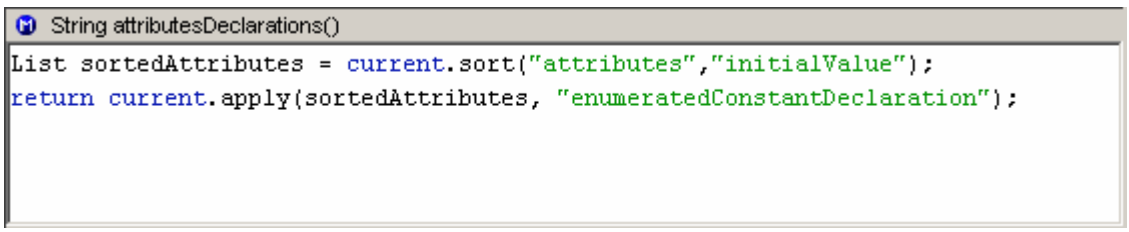


Figure 102 - Macro attributesDeclaration

Note :

The method **sort** enables you to sort a list of objects. This method handles as parameters a list of objects and a script which will be applied to every object in the list.

The result is an instance of *java.util.List* and represents the list of objects sorted by ascending order. For a descending order, you must enter a third parameter (boolean type: **true** for ascending and **false** for descending).

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

You can now test your generation. The creation of scripts has been thus largely facilitated, since it is possible to conceive of them by copying and pasting from existing code for a class.

Generated text (reloaded from Rank.java)

```
public class Rank {

public static final Rank DEVELOPER = new Rank("DEVELOPER");
public static final Rank CONSULTANT = new Rank("CONSULTANT");
public static final Rank MANAGER = new Rank("MANAGER");


    private String id;
    public final int ord;
    private Rank prev;
    private Rank next;
    private static int upperBound = 0;
    private static Rank first = null;
    private static Rank last = null;

    private Rank(String anID) {
        this.id = anID;
        this.ord = upperBound++;
        if (first == null)
            first = this;
        if (last != null) {
            this.prev = last;
            last.next = this;
        }
        last = this;
    }
```

Figure 103 - Rank enumeration generation

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Frameworks

A number of core tasks crop up over and over again. Most developers know that such tasks can and should be pulled into a core framework, built and tested once, and reused across multiple projects. In its simplest form, a framework is simply a body of tried and tested code that is reused in multiple software development projects.

Using generators and frameworks reduce development time while improving delivered software quality -- which means that developers can spend more time concentrating on the business-specific problem at hand rather than on the plumbing code behind it.

For example, the "Safe Type Enumeration" pattern can be pulled into a framework class *AbstractEnumeration*.

```
import java.util.*;

public abstract class AbstractEnumeration {
  private String id;
  public final int ord;
  private AbstractEnumeration prev;
  private AbstractEnumeration next;
  private static int upperBound = 0;
  private static AbstractEnumeration first = null;
  private static AbstractEnumeration last = null;

  protected AbstractEnumeration(String anID) {
       this.id = anID;
       this.ord = upperBound++;
       if (first == null)
            first = this;
       if (last != null) {
            this.prev = last;
            last.next = this;
       }
       last = this;
  }
  public static Iterator elements() {
       return new Iterator() {
            private AbstractEnumeration curr = first;
            public boolean hasNext() {
                 return curr != null;
            }
            public Object next() {
                 AbstractEnumeration c = curr;
                 curr = curr.basicNext();
                 return c;
            }
            public void remove(){
                 throw new UnsupportedOperationException();
            }
       };
  }
  public String toString() {
       return this.id;
  }
  public static int size() {
       return upperBound;
  }
  protected static AbstractEnumeration basicFirst() {
       return first;
  }
  protected static AbstractEnumeration basicLast() {
       return last;
  }
  protected AbstractEnumeration basicPrev() {
       return this.prev;
  }
  protected AbstractEnumeration basicNext() {
       return this.next;
  }
```


This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

```
}
```

Enumeration classes should extend this abstract class.

```
public class Rank extends AbstractEnumeration {

  public static final Rank DEVELOPER= new Rank("DEVELOPER");
  public static final Rank CONSULTANT= new Rank("CONSULTANT");
  public static final Rank MANAGER= new Rank("MANAGER");

  private Rank(String id) {
      super(id);
  }

  public static Rank first() {
      return (Rank) basicFirst();
  }

  public static Rank last() {
      return (Rank) basicLast();
  }

  public Rank next() {
      return (Rank) basicNext();
  }

  public Rank prev() {
      return (Rank) basicPrev();
  }

}
```

Frameworks and generators can add significant value to your project. You can be sure that your framework will grow over time. If your code generator contains only elements depending model, you'll be able to add functionnalities to your generated classes without changing your generator and regenerating but using the extension mechanism of the framework.

When code is generated, the number of errors is considerably reduced. In fact, all typing errors, copy and paste mistakes (very common technique when a generator is not used), omissions, etc. are avoided. These types of errors are not necessarily glaringly obvious, so the risk to leave some in the final code is significant.

However, when using a code generator, if a mistake is done when the script is written, the error is generally reflected on all entities: thus, the detection is fast. A script correction and a complete regeneration make it possible to correct the error.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Using J2SE components into scripts

You can use all Java 2 plateform (J2SE) APIs into MIA-Generation scripts.

The *javax.swing.JOptionPane* class provides many APIs to display dialog boxes and to ask for user inputs. See the documentation of *JOptionPane* for more information.

In the macro *generateAccessors*, we will use the *javax.swing.JOptionPane* class to ask the user if attributes accessors must be generated.

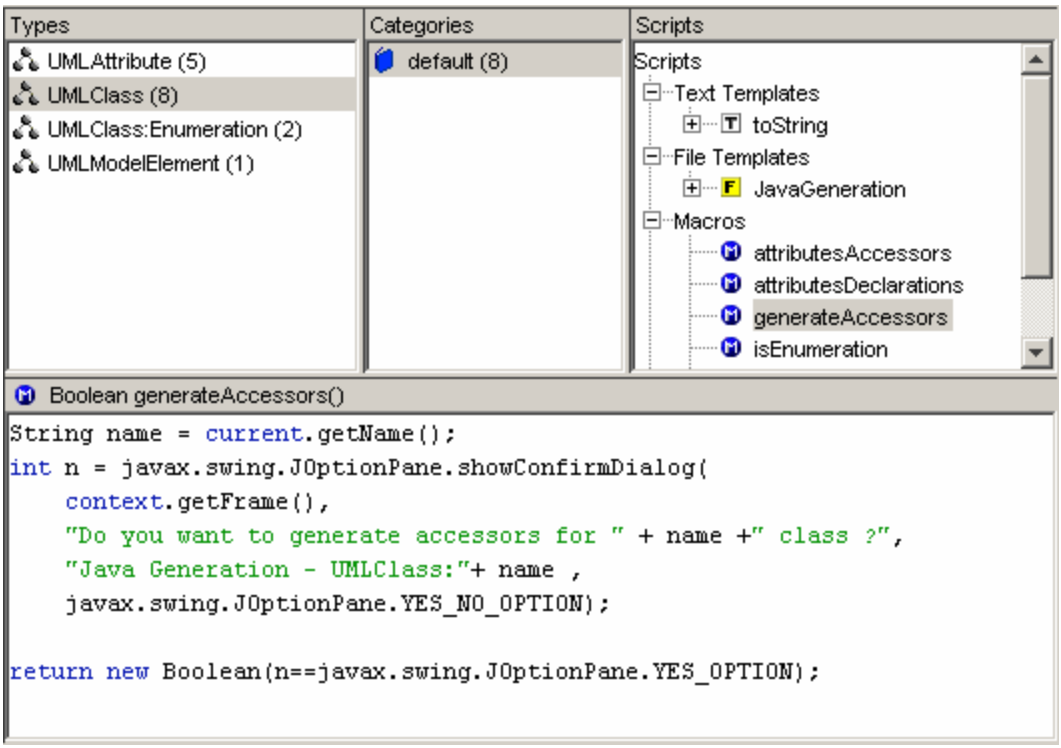


Figure 104 - Macro generateAccessors

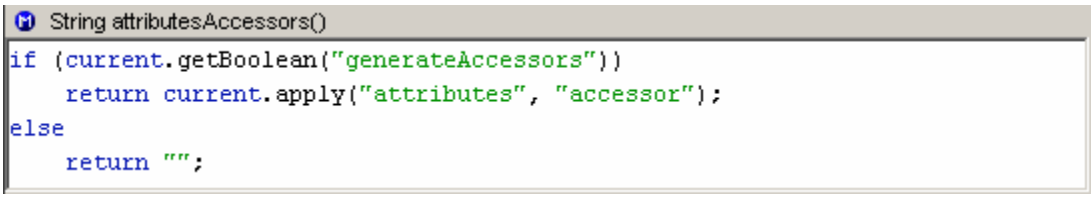


Figure 105 - Macro attributesAccessors

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

A dialog box allow you to generate or not accessors.

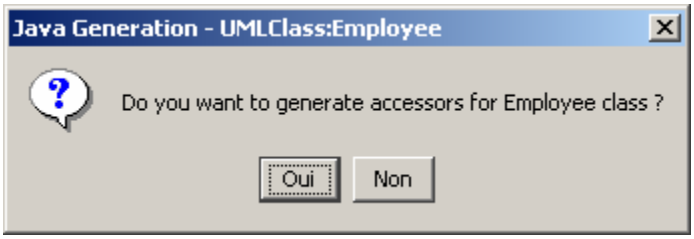


Figure 106 - "generateAccessors" Dialog Box

Note :

In Java™ scripts, you can use any external Java component from MIA-Generation. The **tools/lib** directory in MIA-Generation install directory contains all the user Java components files used by MIA-Generation. All of JAR or CLASS files in this directory are added to MIA-Generation classpath at runtime.

Access to external Java components is done in two steps :

1. Copy your components (JAR file or the directory containing your compiled classes) into the tools/lib directory
2. Use your classes in the scripts : For this, it is necessary to use the complete name, including the name of the package.

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

Context variable

The macro *generateAccessors* uses a variable ***context***.

The variable context allows the definition of variables that can be used in other scripts. This interface provides some helpful services to build generators :

- services giving access to the generation environment, like the generation directory or the project name.
- services allowing to read and write some values into a global variable map.
- services to handle the generation report, if it exists.

Create a new macro *generateAllAccessors* using the context to save the information "Generate Accessors ?" between two calls. This macro uses `cancelGeneration()` method to cancel the generation process.

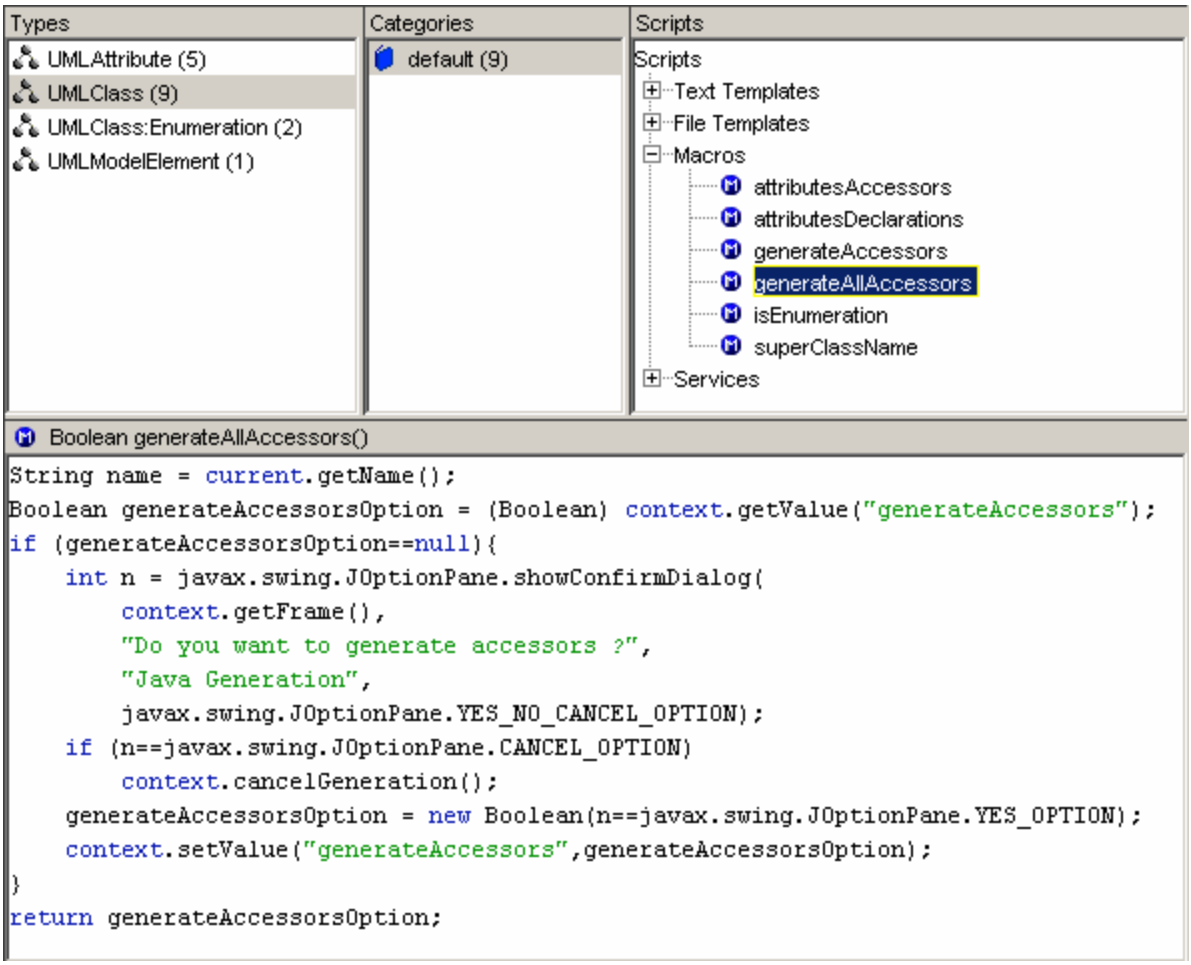


Figure 107 - Macro generateAllAccessors

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

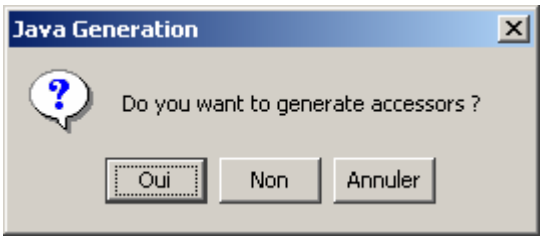
Modifiy the macro *attributesAccessors* to call *generateAccessors* :

String attributesAccessors()

```
if (current.getBoolean("generateAllAccessors"))
    return current.apply("attributes", "accessor");
else
    return "";
```

Figure 108 - Macro attributeAccessors

Launch the scenario *Tutorial*. The dialog box appears only one time.



You can test that context can clear or not between two generations using **Clear Context before generating** checkbox.

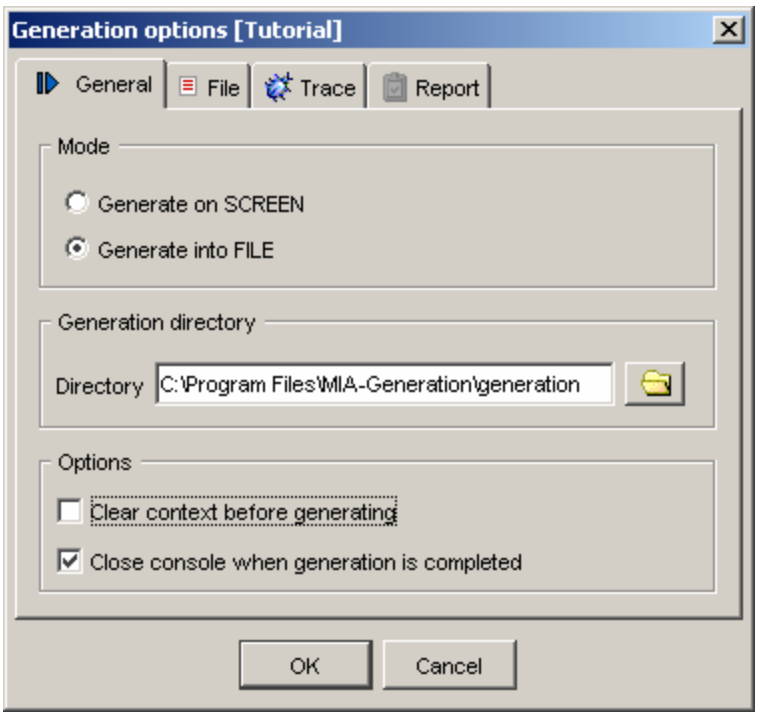


Figure 109 – "Clear Context before generating"

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.

MIA-Generation Javadoc

You can find a Javadoc in the **Help** directory which provides MIA-Generation APIs documention.

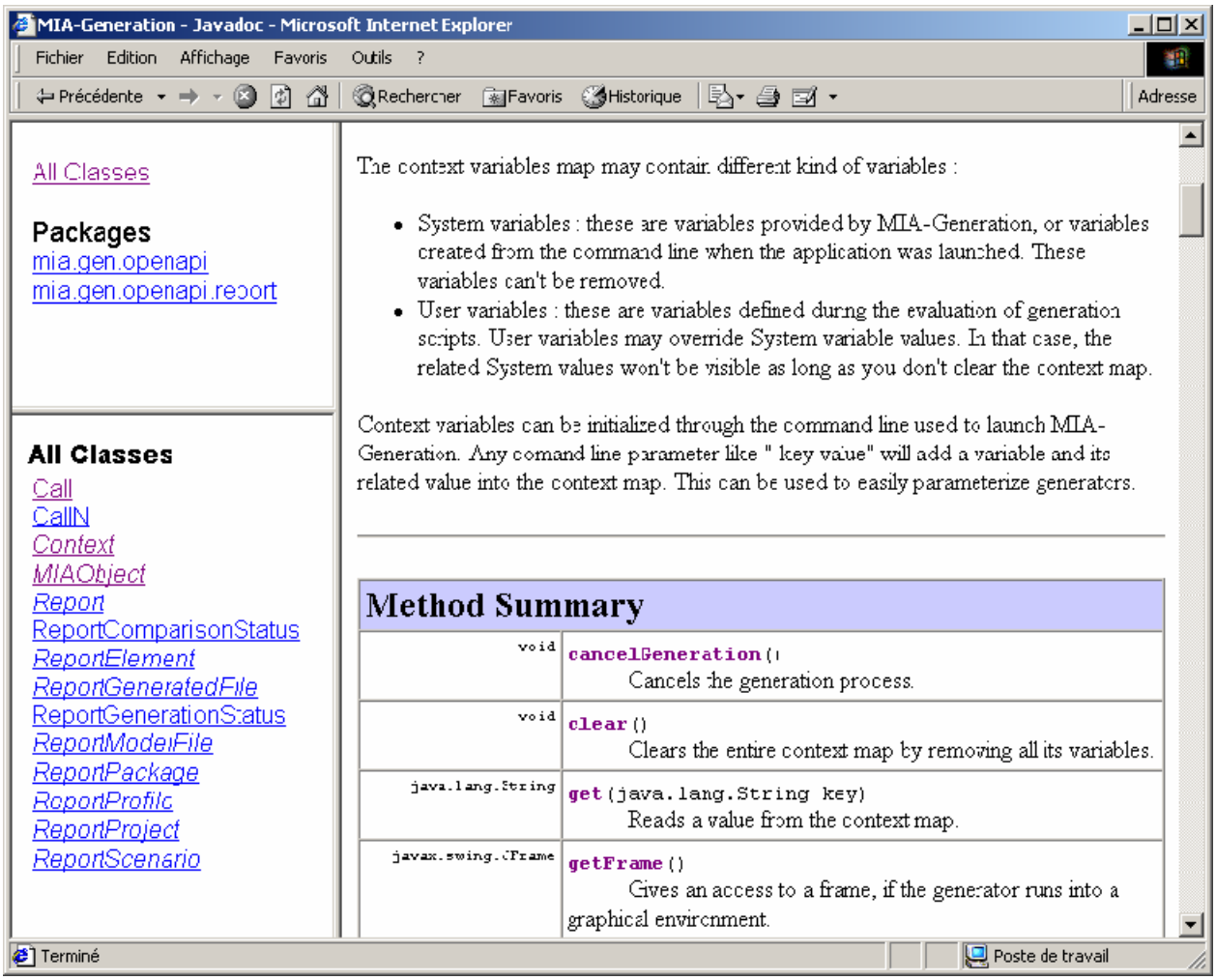

MIA-Generation - Javadoc - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Favoris
Historique
Adresse
All Classes
Packages
mia.gen.openapi
mia.gen.openapi.report
All Classes
Call
CallN
Context
MIAObject
Report
ReportComparisonStatus
ReportElement
ReportGeneratedFile
ReportGenerationStatus
ReportModelFile
ReportPackage
ReportProfile
ReportProject
ReportScenario
The context variables map may contain different kind of variables :
System variables : these are variables provided by MIA-Generation, or variables created from the command line when the application was launched. These variables can't be removed.
User variables : these are variables defined during the evaluation of generation scripts. User variables may override System variable values. In that case, the related System values won't be visible as long as you don't clear the context map.
Context variables can be initialized through the command line used to launch MIA-Generation. Any comand line parameter like " key value" will add a variable and its related value into the context map. This can be used to easily parameterize generators.
Method Summary
void
cancelGeneration()
Cancels the generation process.
void
clear()
Clears the entire context map by removing all its variables.
java.lang.String
get(java.lang.String key)
Reads a value from the context map.
javax.swing.JFrame
getFrame()
Gives an access to a frame, if the generator runs into a graphical environment.
Terminé
Poste de travail

Figure 110 - MIA-Generation Javadoc

This document is the property of Mia-Software.
It cannot be reproduced, even partially, by any means, without prior written permission.